Maps: A Compiler-Managed Memory System for Software-Exposed Architectures

by

Rajeev Barua

B.Tech., Computer Science and Engineering
Indian Institute of Technology, New Delhi, 1992

S.M., Electrical Engineering and Computer Science
Massachusetts Institute of Technology, 1994

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

January 2000

Signature of Author: ______________________________
Department of Electrical Engineering and Computer Science
January 21, 2000

Certified by: ______________________________
Saman Amarasinghe
Assistant Professor of Computer Science and Engineering
Thesis Supervisor

Certified by: ______________________________
Anant Agarwal
Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by: ______________________________
Arthur C. Smith
Chairman, Departmental Graduate Committee

Maps: A Compiler-Managed Memory System for Software-Exposed Architectures

by

Rajeev Barua

Submitted to the Department of Electrical Engineering and Computer Science on January 21, 2000 in partial fulfillment of the requirements for the Degree of Doctor of Philosophy in Electrical Engineering and Computer Science

ABSTRACT

Microprocessors must exploit both instruction-level parallelism (ILP) and memory parallelism for high performance. Sophisticated techniques for ILP have boosted the ability of modern-day microprocessors to exploit ILP when available. Unfortunately, improvements in memory parallelism in microprocessors have lagged behind. This thesis explains why memory parallelism is hard to exploit in microprocessors and advocate *bank-exposed architectures* as an effective way to exploit more memory parallelism. Bank-exposed architectures are a kind of software-exposed architecture: one in which the low-level details of the hardware are visible to the software. In a bank-exposed architecture, the memory banks are visible to the software, enabling the compiler to exploit a high degree of memory parallelism in addition to ILP. Bank-exposed architectures can be employed by general-purpose processors, and by embedded chips, such as those used for digital-signal processing.

This thesis presents Maps, an enabling compiler technology for bank-exposed architectures. Maps solves the problem of *bank-disambiguation*, *i.e.*, how to distribute data in sequential programs among several banks to best exploit memory parallelism, while retaining the ability to disambiguate each data reference to a particular bank. Two methods for bank disambiguation are presented: *equivalence-class unification* and *modulo unrolling*. Taking a sequential program as input, a bank-disambiguation method produces two outputs: first, a distribution of each program object among the memory banks; and second, a bank number for every reference that can be proven to access a single, known bank for that data distribution. Finally, the thesis shows why non-disambiguated accesses are sometimes desirable. Dependences between disambiguated and non-disambiguated accesses are enforced through explicit synchronization and *software serial ordering*.

The MIT Raw machine is an example of a software-exposed architecture. Raw exposes its ILP, memory and communication mechanisms. The Maps system has been implemented in the Raw compiler. Results on Raw using sequential codes demonstrate that using bank disambiguation in addition to ILP improves performance by a factor of 3 to 5 over using ILP alone.

Thesis Advisors: S. Amarasinghe, Assistant Professor, Computer Science & Engineering
A. Agarwal, Professor, Computer Science & Engineering

Dedication

To my wife, Alpana.
Thank you for your love, support and patience!

Acknowledgments

There are several people I would like to thank, the foremost among them are my two advisors, Saman Amarasinghe and Anant Agarwal; and my fellow student, Walter Lee. Saman became my advisor three years ago, but his impact on my thesis has been great. My research benefited tremendously from his enthusiasm, and his willingness to put in many hours on discussions, brainstorming and giving comments. His deep knowledge of compilers helped find innovative solutions to many of the problems we tackled in the Raw compiler. He discovered many of the ideas in this thesis during our many brainstorming sessions. His passion for perfection pushed me to strive for high standards in talks and papers. His help, advice and support have been invaluable in many matters, from writing papers to applying for a job.

Anant Agarwal has been my advisor throughout my years in graduate school. One of the smartest people I know, Anant has the uncanny ability to spot what is important amongst a myriad of possibilities, and state it in one or two sentences. There have been innumerable instances when I have walked into his office, depressed about some experiment not going well; then after hearing me out, he finds a way out, or suggests a new direction – and suddenly, the problem seems more tractable. His feel for what makes for a worthwhile direction of research is unparalleled. His boundless energy, positive attitude and fantastic people skills have made it a pleasure to work with him.

I would like to thank Walter Lee, my collaborator in designing and implementing the Raw compiler. I have very much enjoyed working with Walter. We worked closely on all aspects of the Raw compiler; his ideas have left their mark on virtually every aspect of this thesis. Though Walter was primarily responsible for the back-end space-time scheduler, and I for the front-end memory system, we collaborated extensively on both parts. In this thesis, Walter first suggested mapping each alias equivalence class to a single node yielding equivalence class unification. Walter was invaluable in many brainstorming sessions that ultimately yielded several other ideas in Maps. He was also a co-author on all the papers that led to this thesis. Finally, Walter wrote a great deal of the common infrastructure that went into the Raw compiler.

Martin Rinard and Radu Rugina were key contributors to Maps. Martin first explained to me the power and capabilities of pointer analysis. My early discussions with Martin were invaluable in realizing how pointer analysis might be leveraged in the Raw compiler. Since then, Radu has been extremely helpful in providing his powerful pointer analysis package, as well as customizing it to interface to Raw. He has always made himself available for implementing new functionality in Raw and for bug fixes. I am grateful for his time and efforts. Martin, besides helping in technical matters, has been a friend and an informal advisor; his down-to-earth advice have been a great resource in navigating my job-search and planning my future career.

The concept of software-exposed architectures, and the Raw Machine in particular, was the result of the efforts of several people listed as the authors of [1]. Among these, I would like to thank Elliot Waingold and Michael Taylor for the development of the Raw simulator. This simulator has been used to obtain all the timing results in this thesis. Jonathan Babb helped revamp the Raw compiler directory structure and provided several makefiles to make it easy to run benchmarks. He also provided the Raw benchmark suite [2] which we used to evaluate the Raw compiler. Devabhaktuni Srikrishna wrote the unrolling and renaming passes for the Raw compiler. Matthew Frank was helpful in many related discussions, and he provided several benchmarks. He also gave valuable feedback on my talks and papers. Benjamin Greenwald, who knows more about Unix systems than anyone else I know, bailed us out of infrastructure-related trouble on many occasions. Andras Moritz provided detailed and useful feedback on many of my talks.

My thesis reader, Charles Leiserson, helped me become a better writer. He taught me things about presentation, organization and formatting that I never knew before, and might have never known but for him. His feedback on the thesis helped improve the draft almost beyond recognition. I will never forget his advice; I am sure it will help me in whatever I write in the future.

Krste Asanovic and Arvind have been very helpful in providing feedback and outside perspective on this work. Both discussed my work with me for hours, and helped improve its presentation. Krste helped me understand vector machine compilers better, and their relationship to compilation in Maps.

I thank my parents and sisters for their constant love, support and encouragement. My parents instilled in me the work ethic and integrity that have enabled me to work hard on a doctorate program. I can never repay them for all that they have done for me; I can only give my heartfelt thanks. To my friends in Boston: thank you for your companionship and support.

Finally, I thank my wife, Alpana, for her friendship, love and understanding; she makes life worth looking forward to. She has patiently gone through most of my long, hard years in graduate school; always cheerfully and with love; cheering me when I did well, and making me feel better when things were down. She, more than anyone else, gave me the strength to complete a doctoral program. She has been very understanding in the many instances I have been too busy and unavailable while working on my thesis; I promise to do better in the future!

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Microprocessors over the last two decades have made great strides in their ability to exploit instruction-level parallelism (ILP). During the same period, however, parallelism in accessing primary memory has improved at a much slower rate. Consequently, memory has become a major bottleneck in processor design and threatens to become an even bigger problem. Dick Sites, the chief designer of Alpha microprocessors at the time, warns in his 1996 column titled, “It’s the memory, stupid!” [3] that future performance gains are in jeopardy unless memory performance improves. Figure 1.1 shows the trend over the last two decades [4]. The figure shows that ILP rates have improved from 1 instruction per cycle in the early 1980s, to 4 to 6 instructions per cycle today. Yet, the number of independent words that can be accessed per cycle from primary L1 cache has remained virtually unchanged, from 1 per cycle in the early 1980s, to 1, or at most 2 per cycle today. For example, the Alpha 21364 [5, 6], expected to ship in mid-2000, supports up to 6 instructions and 2 cache accesses per cycle.

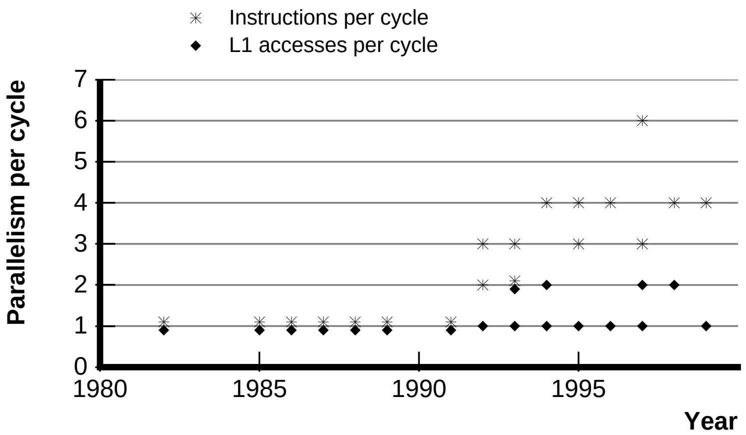


Figure 1.1: ILP and memory parallelism for various microprocessors over the last two decades. Points in the graph represent commercial microprocessors from that year. Each microprocessor contributes two points, one for its maximum number of instructions per cycle (ILP), and one for its maximum number of primary cache (L1) accesses per cycle (memory parallelism). Memory parallelism has remained at 1 or 2 accesses per cycle, while ILP has improved much more.

Even though microprocessors have become increasingly complex over the last decade, the abstraction microprocessors provide to the software has remained the same. Most microprocessors provide the abstraction of a single processing element (PE) accessing a single memory bank. This simple view in software, however, comes at a price: *the overhead of a providing a hardware-based unified view of memory has resulted in memory parallelism remaining stagnant over the last two decades.* Section 1.1 shows why hardware-based unified memory systems limit memory parallelism and prevent scaling to a larger number of banks.

Although it is desirable, for ease of programming, to provide a unified view of memory to the programmer, the hardware need not provide the unified view; instead this thesis explores how the *compiler* can provide a unified view of memory to the programmer on top of distributed memory hardware. The performance of the architecture can be vastly improved if it does not need to provide a unified view of memory. The hardware can, instead, expose its multiple memory banks to the low-level software by eliminating the unified memory hardware. Such architectures with exposed memory banks are called *bank-exposed architectures*. In such architectures, the abstraction presented to low-level software is not that of a monolithic memory; instead load/store instructions target particular memory banks encoded in the software. The user continues to program in a convenient sequential model – sequential programs are desirable since they are easier to write, debug and port to different architectures.

This thesis presents compiler technology that provides a unified view of memory *in the compiler* on top of the distributed memory hardware of bank-exposed machines. Banks are exposed to the low-level software, but not to the programmer. The Maps compiler system presented in this thesis provides a convenient sequential programming model, yet it exploits a high degree of memory parallelism on a bank-exposed architecture.

We show that *bank disambiguation* is the main compiler technique for obtaining good performance on bank-exposed architectures. A memory reference instruction in a program is said to be bank-disambiguated when the compiler guarantees that every dynamic instance of that instruction references the same compile-time-known bank. We show that bank disambiguation allows accesses on bank-exposed architectures to avoid the complex hardware that provides a unified view of memory, and reduces the wire delay incurred by memory references. This thesis presents two methods for bank disambiguation: *modulo unrolling* and *equivalence-class unification*. Bank disambiguation may not disambiguate all accesses: this thesis shows how non-disambiguated accesses can be handled. The major challenge for non-disambiguated accesses is that, for good performance, their long access latencies should be overlapped with computation and other communication as much as possible, while respecting all dependences. This thesis presents a general scheme for overlapping access latencies of non-disambiguated accesses called *software serial ordering*. Two optimizations on software serial ordering, namely *independent epochs* and *updates*, are also presented.

An outline of this chapter follows. Section 1.1 compares the hardware and compiler-directed approaches to providing a unified view of memory to the programmer. It explains why it is difficult to scale hardware-based unified memory, and highlights opportunities available in the compiler to improve scalability and performance. Section 1.2 advocates the bank-exposed class of architectures; such architectures rely on a compiler-directed approach to improve memory performance. Section 1.3 shows why bank disambiguation is the key compiler technology needed for bank-exposed architectures, and outlines our two techniques for bank disambiguation. The section previews results showing performance improvements from using our techniques for bank disambiguation on a particular bank-exposed design, the MIT Raw machine [1]. Results show that using bank disambiguation in addition to ILP improves performance by a factor of 3 to 5 over using ILP alone. Section 1.4 illustrates the challenge in efficiently handling accesses that are not disambiguated by bank disambiguation, and outlines our techniques for handling non-disambiguated accesses. Section 1.5 shows one possible compiler flow for a bank-exposed architecture. Section 1.6 provides an brief overview of the thesis by outlining the scope of our methods, the contributions of the thesis, and the organization of the thesis.

1.1 Unified memory: a hardware vs. compiler approach

This section begins by describing how conventional microprocessors provide a unified view of memory using hardware. This section then explains why delay through the arbitration logic and wire delay make it difficult to scale hardware-based unified memory to a high degree of memory parallelism. Finally, it outlines opportunities available in using the compiler instead of complex hardware to provide a unified view of memory; thus setting the stage for the compiler-directed solution outlined in the rest of this chapter.

Difficulties scaling hardware-based unified memory

This sub-section describes the hardware-based unified memory systems of conventional microprocessors, and discusses how the design prevents scaling to a higher level of memory parallelism. Figure 1.2 shows the typical organization of a hardware-based unified memory system in a conventional microprocessor. Processing elements (PEs) refer to elementary ALUs present on the chip; the memory banks (L1) refer to the multiple banks in the primary level-1 cache. Figure 1.2 does *not* show the complete layout of the chip, only the interface between the processing elements and memory. A unified view of memory is provided by a layer of hardware we call *arbitration logic*. The memory semantics provided by the arbitration logic is that of a single monolithic memory bank accessed sequentially, although the arbitration logic actually connects to multiple memory banks in an effort to exploit memory parallelism. Typically, the arbitration logic consists of routing hardware for requests to and replies from memory banks, as well as a write buffer with associated hardware to enforce memory dependences between outstanding memory requests. To achieve sequential memory semantics, the arbitration logic stores all outstanding requests, routes each request to the bank on which the requested data resides, ensures sequentiality between accesses to the same bank, and routes back results to the requesting processing elements. The results to any one processing element are returned in the order they were requested. Correct sequential semantics are maintained while aiming to overlap requests to different banks as much as possible.

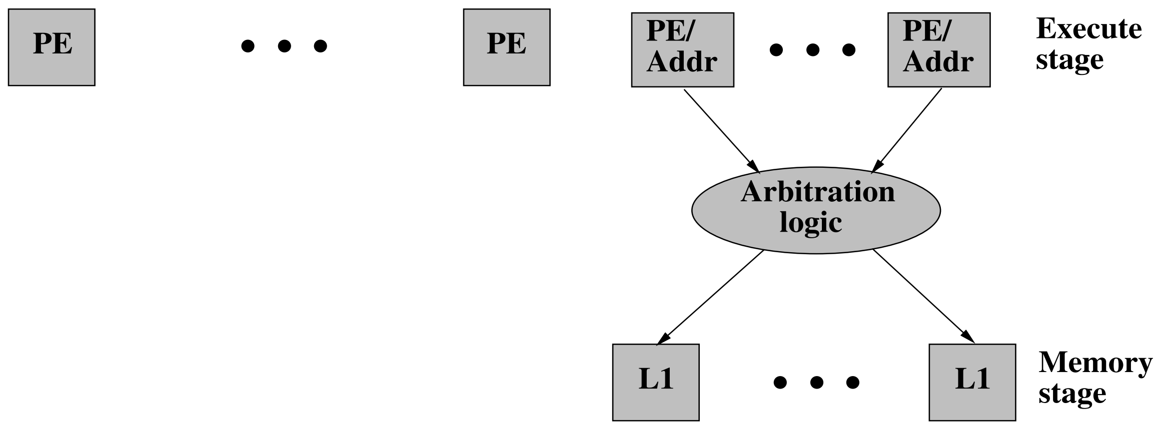


Figure 1.2: Hardware-managed memory system in conventional microprocessors. Processing elements (PE) are of two kinds. Some PEs can compute the effective address of load/store instructions (PE/Addr), and are connected to primary cache banks (L1) through the arbitration logic. Others (PE) are specialized to non-memory instructions alone, and do not interface with the memory system. The number of PE/Addr and L1 are kept small to restrict the complexity of, and delay through, the arbitration logic.

While unified memory hardware simplifies the compiler's task, such hardware has two costs associated with scaling the memory system. First, unified memory hardware does not scale well with the degree of memory parallelism desired. Increasing memory parallelism requires more PEs to be connected to the arbitration logic to issue requests and more memory banks connected to satisfy the requests. Unfortunately, greater connectivity with the arbitration logic increases its delay. Since the arbitration logic delay is a part of the hit time of the primary cache, the delay must be small. To minimize the arbitration logic delay, all commercial microprocessors today limit their memory parallelism to 1 or at most 2 parallel L1-cache accesses per cycle.

A second cost of a unified view of memory in hardware is that it implies the use of long wires to access memory, resulting in poor *on-chip locality*. Long wires result from the use of a single gateway, *i.e.*, the arbitration logic, to access the different banks. Processing elements and cache banks, connected through the arbitration logic, are in general distributed over far-flung areas of the chip, implying long wires. Thus, cache hits, which go through the arbitration logic, may on average traverse half the chip diameter each way. In a billion-transistor, several-gigahertz processor of the future, such a cache hit will become a multi-cycle operation from the wire delay alone. Figure 1.3 illustrates the trend of increasing wire delays by plotting expected cross-chip wire-delay over the next decade[1]. The figure shows that it will take four cycles to cross a chip by 2002 and sixteen cycles by 2007, up from just one cycle in 1997. With increasing wire delay, the primary cache hit time will grow rapidly if the memory system is organized as a monolithic unit accessed from far corners of the chip. Further, the access time within a monolithic cache will also grow as cache sizes grow with VLSI generations.

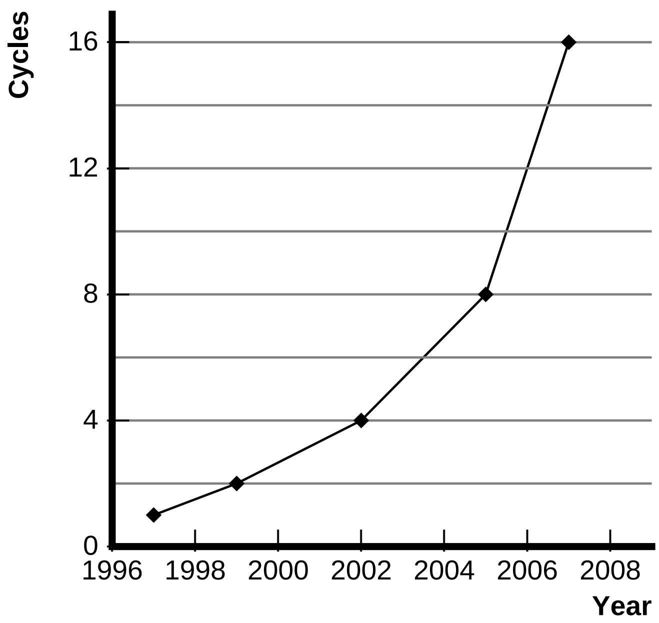


Figure 1.3: Predicted wire delay in cycles for a signal to cross the chip diameter over the coming decade.

Figure 1.4 further illustrates why arbitration logic does not scale with the degree of memory parallelism desired. Let N be the number of PE/Addr in the conventional microprocessor design in figure 1.2; consequently N is also the number of memory instructions that can issue in parallel in one cycle. Figure 1.4(a) shows a sample code with N memory reference instructions, each of which may be either a load or a store. Assume that these N references are issued in parallel to the arbitration logic on a machine with N PE/Addr units. Figure 1.4(b) shows the condition that needs to be hardware-executed in the arbitration logic at run-time before reference $addr_i$ ($i \in [1, N]$) can proceed to its memory bank. The condition shows that for a reference to proceed, it must compare its address with all references issued in the same cycle that are earlier in program order. No check is

[1] Figure 1.3 is derived by combining Matzke's prediction for cross-chip delay versus processor feature size [7] with the Semiconductor Roadmap's prediction for feature size versus year [8]. Matzke assumes that as feature sizes reduce, wires will become thinner, increasing their resistance and hence delay. He also assumes that clock cycle times will reduce with feature size as predicted, thus further increasing the number of cycles for any fixed wire delay.

necessary when both the reference and the previous reference in consideration are loads. A reference proceeds only when all its preceding references in program order that conflict with it have completed[2]. Unfortunately, the implementation of this check in hardware is not scalable with N. In particular, the minimum delay through the logic is $O(\log N)$ by the time all the N references have issued. To see this, consider that for the last reference to proceed, it must perform a binary inclusive *or* of the results of address checks with all $N-1$ previous references; an $O(\log(N-1)) = O(\log N)$ operation. Further, the silicon area of arbitration logic increases with N, reducing scalability; it is possible to achieve $O(\log N)$ delay with $O(N)$ silicon area using a prefix tree scheme [9], or with $O(N^2)$ area using all-to-all checks that improve the delay by a constant factor. In either case, the minimum area is $O(N)$ and the minimum delay is $O(\log N)$; thus the area and delay both increase with N. The wire delay and power consumption also increase with N in any such scheme.

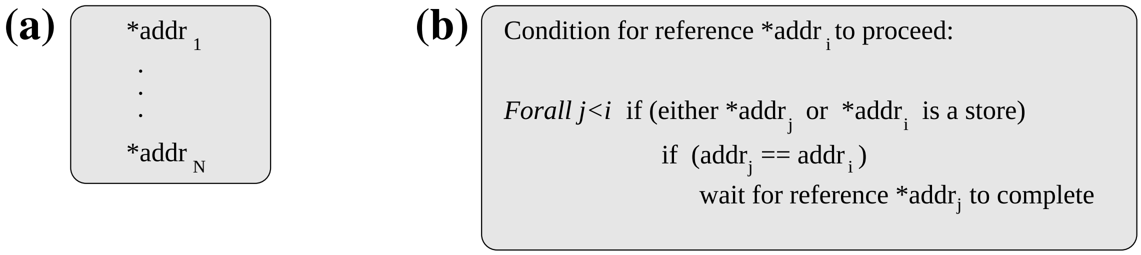


Figure 1.4: Illustration of why arbitration logic does not scale. (a) Code fragment with N memory reference instructions, assumed issued in parallel to the arbitration logic in a conventional microprocessor with N PE/Addr (see figure 1.2). (b) Condition for $addr_i$ ($i \in [1, N]$) to proceed to its memory bank. Executing the condition in hardware requires delay and area that increase with N.

The non-scalable delay and latency of unified memory systems threatens continuing improvements in microprocessor performance. Already, the effects are becoming visible. The primary cache hit time for the Alpha 21264 [6] has increased to 2 cycles, up from 1 cycle for the 21164, because of increased wire delay in bank access. At the same time, the number of concurrent memory accesses allowed has stagnated at 1 or 2 per cycle. Bill Dally of Stanford University recognized the memory-limited nature of today's designs in his 1996 column [10] by saying:

> "Computers in 2006 will be memory, not processor, dominated. Cost will be driven by memory capacity, and performance by memory latency and bandwidth."

[2]Optimizations are possible in certain cases, for example, when a load follows a store in program order, and they refer to the same location. It is possible to implement a hardware scheme in which the load need not wait for the store to complete; instead the load returns the value being stored by the store, without going to memory. Nevertheless, even with such optimizations, the number of checks in figure 1.4(b) does not decrease.

Dick Sites, the chief designer of Alpha chips at the time, takes the same position in his 1996 column [3] by concluding:

> "Over the coming decade, memory subsystem design will be the *only* important design issue for microprocessors."

Opportunities in software

Information discovered by the compiler provides an opportunity to improve microprocessor performance, including memory performance. Compilers have highly detailed information about ILP, memory parallelism and locality that instruction sets of conventional machines fail to take advantage of. For instance, superscalar architectures discover ILP in hardware, while presenting a single-issue instruction-stream interface to software. Compilers often know which instructions can execute in parallel, but there is no way for them to convey this ILP information to the hardware through a superscalar instruction-set architecture (ISA). As a result, superscalar hardware must rediscover ILP information at the cost of considerable area, delay, and power. VLIWs expose their instruction-issue slots to the software, and can exploit somewhat more compiler information. Although VLIWs do not exploit compiler-known memory parallelism information, they do exploit compiler-known ILP information. Interest in VLIWs has resurged recently, fueled by their avoidance of costly ILP-discovering hardware, combined with the increasing sophistication of compilers. Recent VLIW-like processors include the Intel/HP EPIC architecture [11, 12], Transmeta's x86 chip [13], Tensilica's Xtensa embedded CPU [14], and the MAJC chip from Sun Microsystems [15].

Even VLIWs, however, cannot exploit much of the information that compilers can provide, and compilers can potentially discover far more information. In particular, they can optimize for both memory parallelism and on-chip locality. First, compilers can provide information about which memory instructions can execute in parallel. VLIW and superscalar machines cannot exploit memory-parallelism information, as such machines use hardware-based unified-memory systems that do not scale. Second, compilers can provide information about data access patterns. Access-pattern information can be used to derive data layouts and instruction schedules that optimize for on-chip locality. Conventional architectures fail to exploit on-chip locality because they use long wires.

Exposing resources to software

The inability of conventional architectures to use all available compiler information provides an opportunity to improve performance if new kinds of architecture can fully exploit compiler information. One way to exploit more compiler knowledge is to expose more hardware resources to the software. Resources that can be exposed include processing elements, memory banks and the communication network (wires) on chip. Architectures that expose their resources to software are given the generic name of *software-exposed architectures* [1]. VLIWs are partially software-exposed since they expose processing elements through multiple instruction-issue slots. In this thesis we focus on bank-exposed architectures.

1.2 Bank-exposed architectures

This section advocates the bank-exposed class of architecture, a class of architecture that exposes its memory banks to the low-level software. The section explains how bank-exposed architectures enable scaling to a high degree of memory parallelism. Next, examples of bank-exposed designs from the past are presented. The MIT Raw machine, a bank-exposed architecture being designed in our research group, is also presented. Finally, a comparison with multiprocessors is made, showing the differences of bank-exposed microprocessors with multiprocessors that also expose their memory banks.

The minimum architectural feature required to exploit the compiler methods in this thesis is that the architecture be a *bank-exposed architecture*. A bank-exposed architecture is a software-exposed architecture with two defining features: first, several disjoint software-visible address spaces, corresponding to different memory banks; and second, memory references that can be directed at compile-time to particular address spaces. Compile-time resolution of the bank number is the key characteristic of bank-exposed machines. Run-time resolution, in contrast, implies a hardware-based unified memory system.

Figure 1.5 compares the memory system of a bank-exposed architecture with a hardware-based unified memory system. Figure 1.5(a) shows a hardware-based unified memory system. Every access goes through the arbitration logic. Figure 1.5(b) shows a bank-exposed architecture. Figures 1.5(a) and (b) do not show the complete architecture, but only the interface between PEs and memory. In figure 1.5(b), there are two ways to access memory. First, memory reference instructions accessing compile-time-known banks travel over an on-chip communication network, avoiding run-time arbitration. Second, memory instructions to compile-time-unknown banks need run-time arbitration, and hence they go through the slower arbitration logic (not shown). Bank-exposed architectures are advantageous only if most memory instructions reference banks known at compile-time. The compile-time task of discovering a bank number for each reference is called *bank disambiguation*, which is discussed in section 1.3.

A motivating advantage for bank-exposed designs is that they can potentially overcome the two costs of hardware-based unified memory: non-scalable delay through arbitration logic and poor on-chip locality. These advantages are gained, however, only when most memory instructions access compile-time-known banks. The advantages of such accesses to known banks, over accesses in a conventional memory system, are as follows. First, accesses to compile-time-known banks avoid going through the arbitration logic, and so they do not suffer the logic's delay. Consequently, the number of banks can be increased without increasing the cache hit time for local accesses. Second, accesses to compile-time-known banks can exploit on-chip locality, as the compiler can place the memory reference close to the bank where the reference's data resides. In the best case,

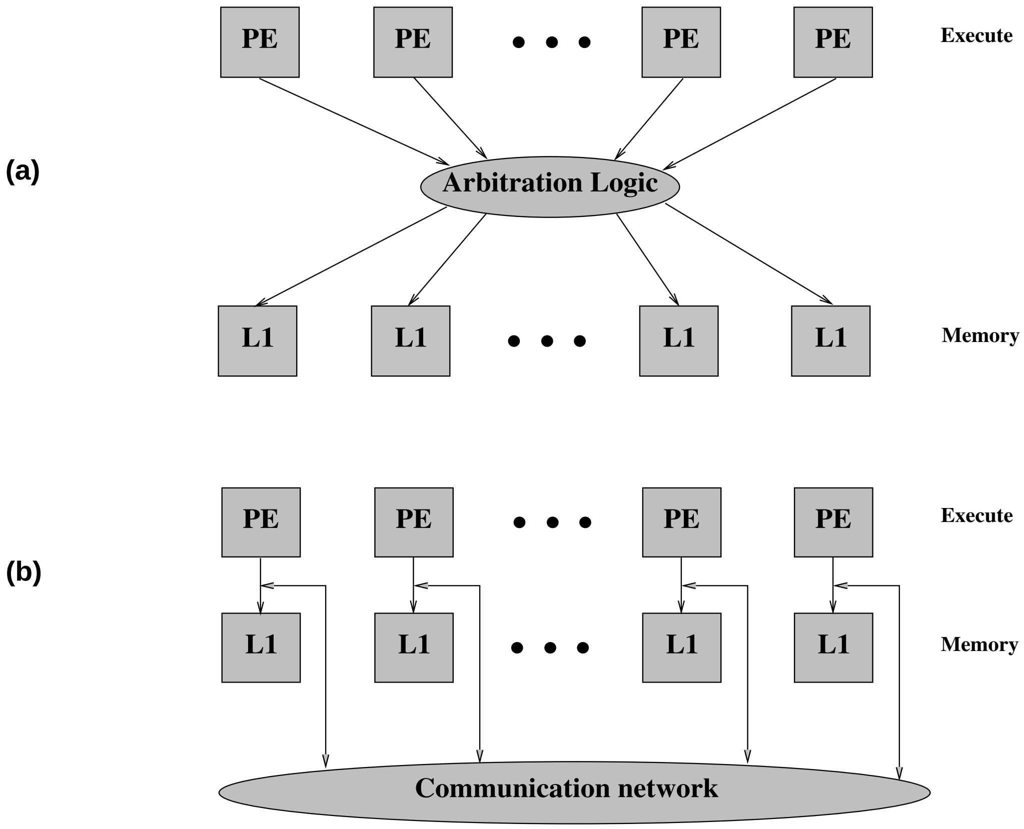


Figure 1.5: Exposed vs. non-exposed memory systems. (a) A hardware-based unified memory system (non-exposed). (b) A bank-exposed architecture. Memory instructions that access a compile-time-known bank complete over an on-chip communication network. Only memory instructions accessing compile-time-unknown banks need to go through the arbitration logic (not shown).

the compiler places the load/store instruction on the PE local to the bank on which the reference's data resides. In this manner, the wire-delay incurred is optimized by the compiler. The L1 cache is split into many small and independently accessed banks, instead of one or two large banks as in a conventional memory system. Accesses to known banks need only go to a small, nearby bank with low access delay, instead of a larger, far-away bank with higher access delay.

We revisit figure 1.4 to see how bank-exposed architectures eliminate the cost of arbitration logic. Figure 1.4(a) shows a code fragment with N memory instructions; figure 1.4(b) shows the condition that the arbitration logic needs to evaluate before it can proceed with issuing the ith reference. Bank-exposed architectures, in contrast, avoid the arbitration logic and its delay for memory instructions to compile-time-known banks. The reason why the arbitration logic is avoided is that accesses proven by the compiler to go to different banks cannot be dependent; hence they need not be checked for dependence at run-time. In addition, accesses known to go to the same bank are serialized at the bank anyway, so checks provide little benefit and are not done. Consequently, accesses to compile-time-known banks require no run-time arbitration whatsoever – the $O(\log N$ delay through arbitration logic is thus avoided, allowing N to be increased to many banks without increasing the cache hit time.

Examples of bank-exposed architectures

Bank-exposed architectures are not a new idea. Bank-exposed general-purpose microprocessors date back to at least 1983 when Josh Fisher proposed the ELI-512 VLIW machine [16]. The ELI-512 is an unusual VLIW that uses a point-to-point network, rather than a bus, to connect its different processing elements, each having its own memory bank. The ELI-512 supports two ways of accessing memory. Accesses to compile-time-known banks use a "front door" to memory, while accesses to compile-time-unknown banks use a slower "back door" to memory. The iWarp machine [17] is another bank-exposed design: it exposes not only memory, but also its processing elements and network. Digital-signal processing (DSP) chips, such as the Motorola DSP56000 family [18], have also used bank-exposed designs. DSP chips usually have 2 to 3 software-exposed banks, called X, Y, and Z memories. DSP chips usually provide no arbitration logic; all accesses must be to compile-time-known banks. Shortcomings in bank-exposed compilers have meant that, even today, most DSP chips are hand programmed.

The MIT Raw machine [1], being developed in our research group at MIT, is the latest in the line of bank-exposed general-purpose designs. The methods in this thesis apply, however, to any bank-exposed design. The Raw machine exposes not only its memory, but also its processing elements and networks. The Raw architecture is shown in figure 1.6. A Raw architecture consists of a 2-dimensional mesh of tiles. Each tile is composed of a processing element and a cache memory bank. A switch is provided on each tile to communicate with other tiles. Two communication networks connect the tiles: the *static network* and the *dynamic network*. The static network is a fast register-level network. Accesses to compile-time-known banks complete over the static network, or are local from a PE to its local bank. The dynamic network is a slower memory-level network that replaces the arbitration logic. Accesses to compile-time-unknown banks complete over the dynamic network. Details of the Raw architecture, along with the reasons for using two networks, are presented in section 2.4.

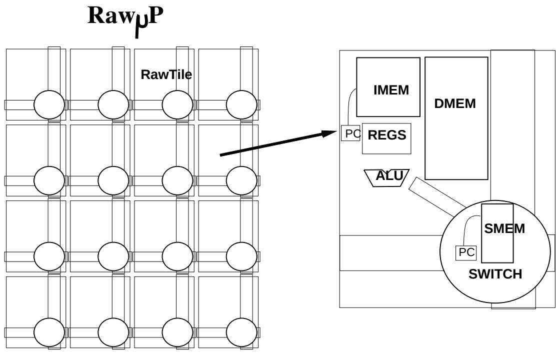


Figure 1.6: A Raw microprocessor is a mesh of tiles, each with a processing element, some memory and a switch. The processing element contains both registers and an ALU. The processing element interfaces with its local instruction memory, local data memory and local switch. The switch contains its own instruction memory.

For any bank-exposed architecture, as in figure 1.5(b), the term *tile* is naturally defined as a memory bank with all its associated functional units. The Raw machine has only one processing element per bank, and hence a tile is a single processing element with its associated memory bank.

Comparison with multiprocessors

Distributed memory multiprocessors, such as Flash [19], Alewife [20] and IBM's SP2 [21], have multiple memory banks exposed to the software. To date, however, the relatively long remote-access latencies of multiprocessors have restricted them to exploiting coarse-grain parallelism rather than ILP.

The reason for the slow access times on multiprocessors is *their lack of hardware support for static scheduling* of memory accesses. Memory accesses can be statically (at compile-time) scheduled if their access patterns are known and the architecture provides a way of statically ordering memory accesses on each bank. Statically ordered references on each bank provide dependence enforcement without expensive overheads. An architecture can provide references that are statically ordered in two ways. Either the architecture guarantees a predictable access time to every bank, or it provides a network that guarantees that messages are delivered in the order specified at compile-time. The Raw architecture takes the second approach by providing the compiler-routed static network. Predictable ordering of messages reduces the overheads of run-time routing. In multiprocessors, the inherent unpredictability in the arrival order and timing of messages requires expensive reception mechanisms such as polling or interrupts, and expensive run-time congestion-control schemes that arise due to the unpredictable ordering of messages. The consequently faster communication on bank-exposed architectures enables their compilers to focus on ILP, instead of coarse-grained parallelism.

Compilers for multiprocessors aim to discover coarse-grain parallelism from sequential programs. Unfortunately, discovering coarse-grained parallelism in the compiler involves complex whole-program analysis, which has thus far proven successful only for dense matrix applications [22, 23]. An alternative to automatic parallelization for multiprocessors is programming them using an explicitly parallel language. This approach, however, has not found widespread acceptance outside a small group of programmers, mostly in the scientific community. Thus, while the ideas in this thesis apply equally to parallel programs, we focus on sequential programs alone.

1.3 Bank disambiguation

This section explains bank disambiguation, and shows why it is the central challenge for extracting good performance from bank-exposed architectures. Bank disambiguation enables memory instructions to go to compile-time-known banks. Bank disambiguation is motivated, and the quality criteria required for bank disambiguation are explained. An important benefit of bank disambiguation, static scheduling of memory instructions, is illustrated. Our two methods for bank-disambiguation, modulo unrolling and equivalence-class unification are outlined. Finally, results demonstrating performance improvements from using our bank disambiguation methods are presented.

Motivation and definition

Bank disambiguation is motivated by the existence of two ways to access memory on a bank-exposed architecture. First, memory instructions that access a compile-time-known bank avoid arbitration logic, using a communication network for access instead. Second, memory instructions that access a compile-time-unknown bank must use the slower arbitration logic. Accesses to compile-time-known banks are preferable as they avoid non-scalable run-time arbitration and exploit on-chip locality. Hence, the compiler aims to find a fixed bank number, known at compile-time, for as many memory instructions as possible.

Definition 1.1 *A particular load or store memory-reference instruction in the program is said to be* ***bank disambiguated*** *to a particular bank if the instruction accesses the same compile-time predictable bank in every dynamic instance.*

Success in bank disambiguation critically depends on the data distributions used. The methods in this thesis carefully choose data distributions so that bank disambiguation is possible for most accesses, resulting in a high degree of memory parallelism. Poorly selected data distributions may imply that most memory instructions in the program go to different banks in different dynamic instances, thus failing disambiguation, even if the distribution provides memory parallelism.

Quality criteria

Not all bank disambiguation methods are of equal quality: some schemes are better than others. Specifically, bank disambiguation needs to be achieved while optimizing for two factors:

- **Memory parallelism** The bank disambiguation scheme should achieve a high degree of memory parallelism. It is easy to bank-disambiguate without memory parallelism: all program data can be allocated to a single bank, thus trivially disambiguating all references to that bank. Better bank disambiguation methods allocate data to several tiles so that different memory instructions can be issued in parallel, yet most memory instructions go to a fixed bank predictable at compile-time.

- **Code size** Some bank disambiguation schemes require program transformations, such as loop unrolling, that increase the code size of the program. Good disambiguation schemes keep code growth to a minimum. The modulo unrolling example presented later in this section shows that a naive scheme for bank disambiguation involves fully unrolling loops. Full unrolling is expensive, however, if the range of loop-bounds is large, and is not possible for unknown loop bounds. The methods in this thesis keep the code growth bounded by a constant factor, independent of the loop bounds.

Bank disambiguation aids static scheduling

A major advantage of bank disambiguation is that it enables the static scheduling of memory references. Static scheduling of memory references requires that the hardware provide a way of statically ordering memory accesses on each bank, either using banks having compile-time-guaranteed latencies, or by providing a compiler-routed network with compiler-specified ordering of messages. Even with hardware that allows static ordering of memory references, without bank disambiguation the banks accessed for each reference are not known, making it impossible to guarantee the ordering of accesses on each bank, thereby making static scheduling impossible. Thus, bank disambiguation is a necessary for static scheduling.

Further, bank disambiguation is also sufficient for static scheduling if the assignment of memory instructions to PEs is under compiler control. A compile-time-known PE for a memory instruction implies that a compiler-routed message can be used from the memory instruction to its bank. Alternatively, if the hardware provides a guaranteed latency to each bank, knowing the PE for the memory instruction provides a guaranteed latency for the entire memory access from PE to the bank. Either way, a compiler-time-known PE assignment for each memory instruction, along with bank disambiguation, is sufficient for static scheduling.

Modulo unrolling

To understand bank disambiguation, consider figure 1.7. The figure shows an example of how bank disambiguation is done using modulo unrolling, one of the two bank disambiguation schemes presented in this thesis. Figure 1.7(a) shows the code fragment forming the compiler input, consisting of a simple **for** loop containing a single array reference $A[i]$. The array $A[]$ is assumed to range from 0 to 99. The array $A[]$ is shown distributed among 4 memory banks using low-order interleaving[3].

Now, suppose we want to bank-disambiguate the $A[i]$ memory instruction on a bank-exposed architecture with 4 banks. For disambiguation, an array reference instruction must access the same bank in every dynamic instance. If the array $A[]$ is distributed in any way at all, then by the definition of distribution, the different dynamic instances of $A[i]$ go to different banks in different iterations, thus failing disambiguation. For example, assume that array $A[]$ is distributed using low-order interleaving, as depicted in figure 1.7(a). The $A[i]$ reference accesses all 4 banks for different values of i, as depicted by the arrows from the reference to all 4 banks. The $A[i]$ reference has the bank-access pattern of 0, 1, 2, 3, 0, 1, 2, 3, ...in successive loop iterations. Since the reference instruction goes to more than one bank, it fails disambiguation.

A naive way to attain bank disambiguation and memory parallelism is to *fully unroll* the loop. Figure 1.7(b) shows the sample loop in figure 1.7(a) fully unrolled, *i.e.*, unrolled by a factor of 100. Full unrolling makes all the array-reference indices constant in the unrolled code, and hence all the array references in the unrolled loop are trivially disambiguated to the bank holding the known array element. For example, the reference $A[99] = \ldots$ is disambiguated to bank 99 mod 4 = 3. Full unrolling provides disambiguation with memory parallelism, but full unrolling is prohibitively expensive in terms of code-size increase. Full unrolling is not even possible for compile-time-unknown loop bounds. Consequently, the Maps system never uses full unrolling.

Fortunately, in this case, there is a way to attain both memory parallelism and bank disambiguation without a huge increase in code size. The solution involves transforming the program using loop unrolling. Figure 1.7(c) shows the result of unrolling the code in figure 1.7(a) by a factor of 4. Now, each access always refers to elements on the same memory bank. Specifically, $A[i]$ always refers to tile 0, $A[i+1]$ to tile 1, $A[i+2]$ to tile 2, and $A[i+3]$ to tile 3. Therefore, all 4 new references are bank disambiguated. Furthermore, the 4 references in figure 1.7(c) can proceed in parallel, thus providing memory parallelism.

---

[3]Low-order interleaving is a data layout in which array elements are interleaved among the N different banks in a round-robin manner, beginning at bank 0. That is, for array $A[]$, element $A[i]$ is allocated on bank i mod N. The array A[] is thus broken up into N sub-arrays, $A_0[]$ to $A_{N-1}[]$, such that $A[i]$ maps to $A_{i \bmod N}[i \text{ div } N]$.

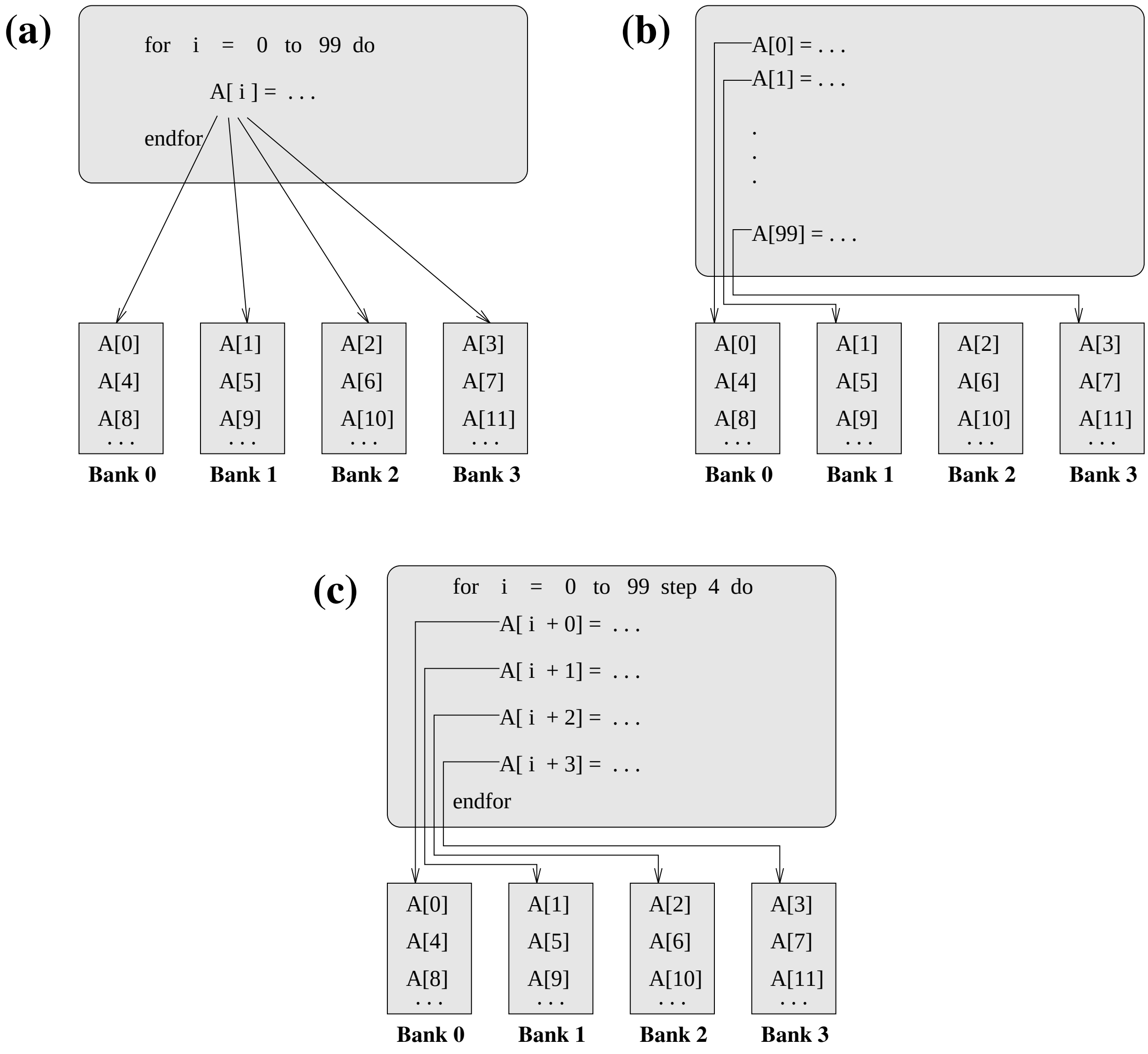


Figure 1.7: Example of modulo unrolling. (a) Original code. Array A is low-order interleaved on a 4-bank bank-exposed machine. The $A[i]$ memory reference instruction goes to different banks for different values of i. (b) Code after full unrolling. Disambiguation is attained, *i.e.*, each reference goes to a single bank, but code size is huge. (c) Code after unrolling by factor 4. Disambiguation is attained with limited code size increase.

The program transformation in figure 1.7 is not a one-instance special case. Rather, it is a specific application of a fully automated general technique for bank disambiguation called *modulo unrolling*. Presented in this thesis, modulo unrolling is a technique that provably disambiguates any array reference in a nested loop if the indices of the array reference are affine functions[4] of the index variables of the enclosing loops. The modulo unrolling transformation provides both bank disambiguation and memory parallelism. Chapter 4 presents details on how modulo unrolling is automated. Affine array accesses are common in scientific codes and some multimedia codes, and are present even in some irregular programs. Modulo unrolling provides a significant performance improvement for such programs.

Equivalence-class unification

Equivalence-class unification (ECU) is the second of the two techniques for bank disambiguation presented in this thesis. ECU is applicable in all cases that modulo unrolling is not applicable; *i.e.*, ECU aims to disambiguate all accesses other than affine array accesses. ECU handles all kinds of accesses in an integrated framework, including non-affine array accesses, pointer dereferences, structure references, and heap references, in programs with arbitrary memory-aliasing.

Figure 1.8 demonstrates ECU through an example. Figure 1.8(a) shows the original code input to the compiler. There are three integers – a, b and c – and two pointer references – $*p$ and $*q$. The value of boolean variable *cond* is assumed to be unknown at compile-time. Figure 1.8(b) shows the code and data after ECU on a 2-banked bank-exposed machine. In order to bank-disambiguate the $*p$ and $*q$ references, ECU places a and b on bank 0, and c on bank 1. With this data allocation, both references go to only one bank, irrespective of the run-time value of *cond*, thus bank disambiguating both references.

Chapter 3 shows how the ECU method can be generalized to handle any input program, and it describes the general method. The ECU method aims for memory parallelism across data objects, rather than within. Arrays are allocated to a single tile; structures, however, are distributed as ECU considers structure fields as individual data objects. A drawback of ECU is that it does not exploit memory parallelism within arrays and array-like heap-allocated memory blocks accessed by non-affine accesses. Modulo unrolling overrules ECU for arrays accessed primarily by affine functions, so arrays are placed on one node only if they are primarily accessed by non-affine accesses.

[4] An *affine function* of a set of variables is defined as a linear combination of those variables, plus a constant. As an example, given i and j as enclosing loop variables, $A[i + 2j + 3][2j]$ is an affine access, but $A[i * j + 4]$ is not.

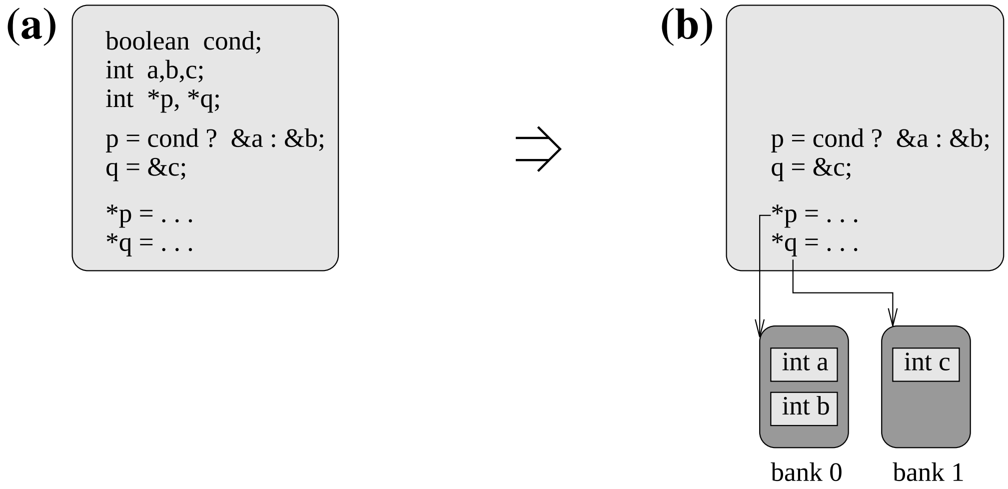


Figure 1.8: Example showing equivalence-class unification (ECU). (a) Original code. (b) Code and data after ECU on a 2-banked bank-exposed machine. The data declarations are not repeated. With integers a, b and c allocated as shown, both pointer references each go to only one bank, enabling disambiguation.

Results

To demonstrate the effectiveness of our bank disambiguation methods, we now preview some results. Maps, including techniques for bank disambiguation and for handling non-disambiguated accesses, has been implemented in a SUIF-based compiler [24] for the Raw machine, called Rawcc. The Maps system is a part of the Raw project's common infrastructure. The Maps infrastructure has been used by at least 6 members of the Raw group. Rawcc accepts as input, ordinary sequential programs with no special user-directives or pragmas. Rawcc automatically detects and exploits both ILP and memory parallelism. Evaluation is performed on a cycle-accurate simulator of the Raw microprocessor.

Figure 1.9 compares the speedup for various programs on a 32-tile Raw machine for two cases: a compiler using ILP alone versus a compiler using ILP augmented with the bank disambiguation methods in this thesis. The baseline for both strategies is the sequential program running on one tile with a speedup of 1. In both sets of numbers, ILP is exploited by compiler-discovery, as in VLIW machines. Our techniques for extracting ILP are described in [25]. The *ILP alone* numbers use 32 PEs, but only one memory bank, as they have no way of disambiguating memory. The *ILP + bank disambiguation* numbers use 32 PE-memory bank pairs. The benefit from bank disambiguation is the improvement in performance of the *ILP + bank disambiguation* numbers over the *ILP alone* numbers. Figure 1.9 demonstrates that using bank disambiguation improves performance over using ILP alone by a factor of 3 to 5 for a broad spectrum of programs[5].

[5] Adpcm, SHA and fppp-kernel are exceptions; see chapter 9 for reasons why.

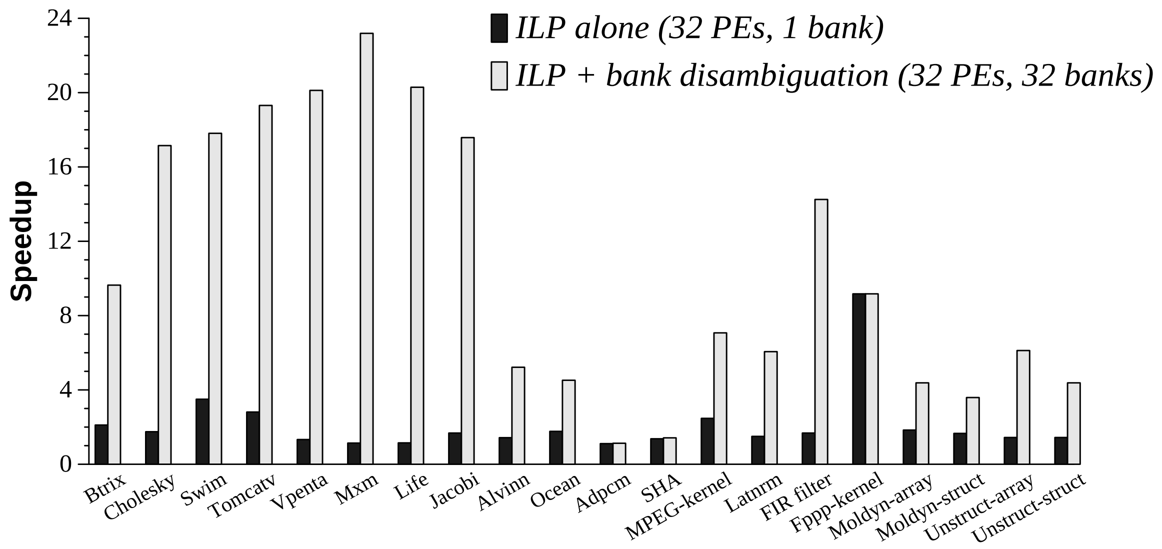


Figure 1.9: Benefits of bank disambiguation. Comparison of 32-tile speedups using instruction-level parallelism (ILP) alone, versus ILP combined with our bank disambiguation techniques.

The results show that ILP alone is not enough; exploiting memory parallelism dramatically improves performance over using ILP alone. This result has significance beyond Raw. Although this thesis makes no direct comparison with machines other than Raw, the configuration for the *ILP alone* numbers approximates a conventional microprocessor: it has a large number of PEs (32) and a monolithic L1 cache that supports 1 memory access per cycle. The results argue that using a bank-exposed architecture, coupled with bank disambiguation, improves performance over using a conventional architecture with a hardware-based unified memory system.

1.4 Non-disambiguated accesses

This section outlines how non-disambiguated accesses are handled efficiently in the Maps compiler system. First, the need of non-disambiguated accesses is motivated. Second, the challenges in efficiently handling non-disambiguated accesses are illustrated. Finally, methods for handling non-disambiguated accesses efficiently in Maps are presented. A baseline method for handling non-disambiguated accesses called *software serial ordering* is presented; an optimization called independent epochs, applicable in certain cases, is also outlined.

Motivation

Although ECU alone can provably disambiguate all memory reference instructions, there are cases where it is not desirable to disambiguate all references. Figure 1.10(a) illustrates one such situation. The figure shows two references, $*p$ and $*q$. The values of variables x and y are assumed to be unknown at compile-time. The pointers p and q point to unknown locations within the array $A[]$ and potentially alias to the same location when $x = y$. Assume that the array $A[]$ is accessed by only affine accesses in the rest of the program and that the affine accesses make up the performance-critical portions of the code. Strict application of ECU would place the array on one tile because of the two non-affine pointer accesses, destroying performance in the performance-critical portions that have affine accesses. Rather than using ECU, it might be better to keep array $A[]$ distributed, use modulo unrolling in the performance critical portions, and use slow non-disambiguated accesses for the $*p$ and $*q$ references. Therefore, an efficient way to deal with non-disambiguated (dynamic)[6] accesses is needed. Amdahl's law predicts that even a small fraction of the program running slowly can significantly degrade the overall performance. Other scenarios where dynamic accesses are helpful are outlined in chapter 5.

Challenges in handling non-disambiguated accesses

To efficiently implement dynamic accesses, their comparatively long access latencies should be overlapped with each other and with computation as much as possible, while respecting memory dependences. If the arbitration logic is hardware-managed, as in most microprocessors, the task of overlapping latencies is done by the arbitration logic hardware. Although sophisticated write-buffers aim to overlap as much latency as possible, the scalability of arbitration logic is limited. Consequently, the Raw machine investigates the use of a scalable, distributed and point-to-point dynamic network in place of the arbitration logic. The lack of run-time arbitration in such networks means that the task of providing correct memory semantics while overlapping latencies is left to the compiler.

Figure 1.10(b) depicts complete serialization, the most obvious way of implementing the code in figure 1.10(a) using dynamic accesses on Raw. Serialization ensures that the possible dependence between the $*p$ and $*q$ references is satisfied. Each dynamic store consists of a non-disambiguated request message to a bank that is unknown at compile-time, followed by an acknowledgment message to the PE scheduled to receive the acknowledgment. In complete serialization, the $*q$ access does not start until the $*p$ access receives its reply. The long latencies of dynamic accesses imply that complete serialization is expensive, as complete serialization fails to overlap the dynamic latencies. Overlapping latencies is tricky, because there are no timing guarantees on a distributed dynamic network, such as on Raw. Simply issuing the requests from different banks in

[6]From this point on, the terms "non-disambiguated" and "dynamic" are used interchangeably. Non-disambiguated accesses resolve their bank numbers dynamically, *i.e.*, at run-time, hence the term dynamic. The dynamic network on Raw is used to complete dynamic accesses.

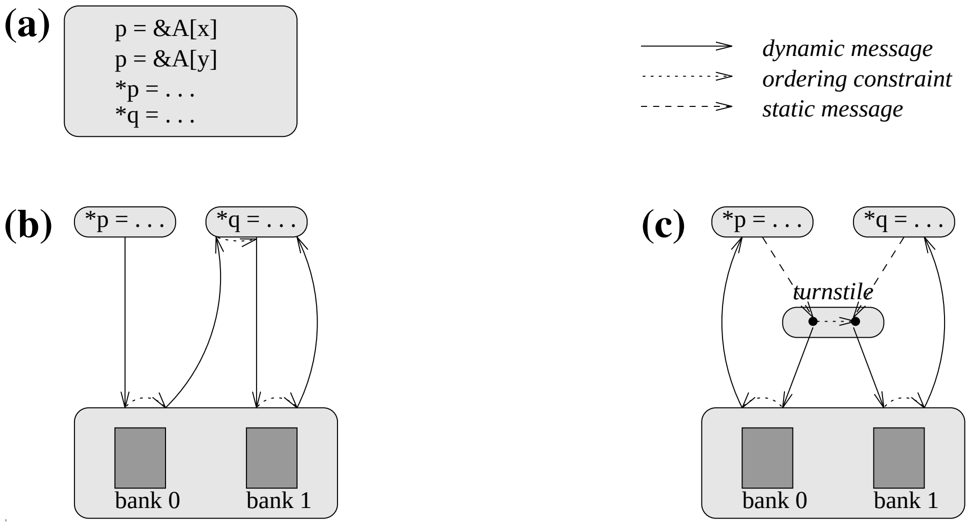


Figure 1.10: Example showing software serial ordering (SSO). (a) Input code. Variables x and y have values unknown at compile-time. (b) Possible dependence enforced through complete serialization. The actual locations accessed are unknown at compile-time. To enforce the possible dependence, the $*q$ reference does not proceed until the $*p$ reference is complete. (c) Dependence enforced through software serial ordering. The only serialization is at the turnstile node. Much of the dynamic latencies are overlapped.

program order using explicit synchronization may violate correctness. Request messages might arrive at a memory bank out of order, even if they are issued in order from different banks.

Software serial ordering

This thesis proposes a method called *software serial ordering* (SSO) to overlap access latencies on distributed dynamic networks while maintaining dependences. SSO relies upon the *in-order, wire-like* property between every pair of tiles, offered by many dynamic networks, including the dynamic network on Raw. This property states that if two or more messages are sent between the same source and destination, they appear at the destination tile in the same order as they were launched at the source.

Figure 1.10(c) shows the code in figure 1.10(a) implemented using SSO on a Raw-like distributed dynamic network. SSO involves serializing accesses at a special *turnstile* node, which may be placed on any of the PEs. The $*p$ and $*q$ references make requests to the turnstile node using statically scheduled messages on the static network. The requests are issued in order, from the turnstile to the memory banks, on the dynamic network. At the destination memory bank, whose number is resolved only at run-time, the access is performed and a store-acknowledgment message is sent back to a compile-time known PE, different for different acknowledgments. Correctness is ensured, because request messages from the turnstile to any one memory bank arrive in program order; in-order delivery follows from the pair-wise in-order property of the network. Hence, if the two references, $*p$ and $*q$, actually alias to the same location at run-time, the request messages arrive at the memory bank containing the location in program order. Finally, when all the acknowledgments are received, the control moves to the next scheduling unit of code after the current scheduling unit. A scheduling unit is the code-granularity at which the Raw compiler performs instruction scheduling; it is defined more precisely in section 1.5.

SSO is slower than ECU. The $*p$ and $*q$ references are faster if array $A[]$ is placed on one node. SSO is slower, because the accesses in SSO are serialized, just as in the ECU case, and they also suffer from dynamic overhead. The advantage of SSO is not in the portion of code it is used. Rather, SSO may allow other portions of code elsewhere in the program to run faster, because, unlike ECU, SSO uses distributed arrays. Consequently, other parts of the program can use modulo unrolling if affine accesses are present, improving the performance of those accesses. SSO has been implemented in the Rawcc compiler, and results are presented in chapter 9.

Independent epochs

It is possible, in certain cases, to implement dynamic accesses without the turnstile's serialization. If the compiler can prove that accesses to a set of data objects in a region of code are all dynamic and are all independent, then the accesses proceed in parallel without a turnstile. Such a region of code is called an *independent epoch*. A trivial example of an independent epoch is a region whose dynamic accesses to a set of data objects are all loads. Otherwise, independent accesses are found using dependence analysis [26]. Dependence analysis is aided by pointer analysis, array-index analysis, and dataflow analysis.

Figure 1.11 shows an example of an independent epoch. Figure 1.11(a) shows the initial code. The code is very similar to that in figure 1.10(a), except that the pointers p and q point to consecutive unknown locations in array $A[]$ instead of any two unknown locations. Dependence analysis reveals that the $*p$ and $*q$ references are always independent, irrespective of the value of variable x. Hence, the two references are issued in parallel without a turnstile. Parallel dynamic accesses are depicted in figure 1.11(b). Turnstile serialization is avoided because the references cannot alias at run-time.

One additional step must be taken to ensure correctness for independent epochs: memory barriers must be placed before and after each independent epoch region. A memory barrier is a construct that guarantees that all outstanding memory requests up to that point have been completed. Memory barriers are needed before and after an independent epoch to isolate the epoch from the accesses elsewhere in the program. Without isolation, an access from inside the epoch could be incorrectly reordered at run-time with a dependent access outside the epoch. Implementation of memory barriers is discussed in section 5.5.

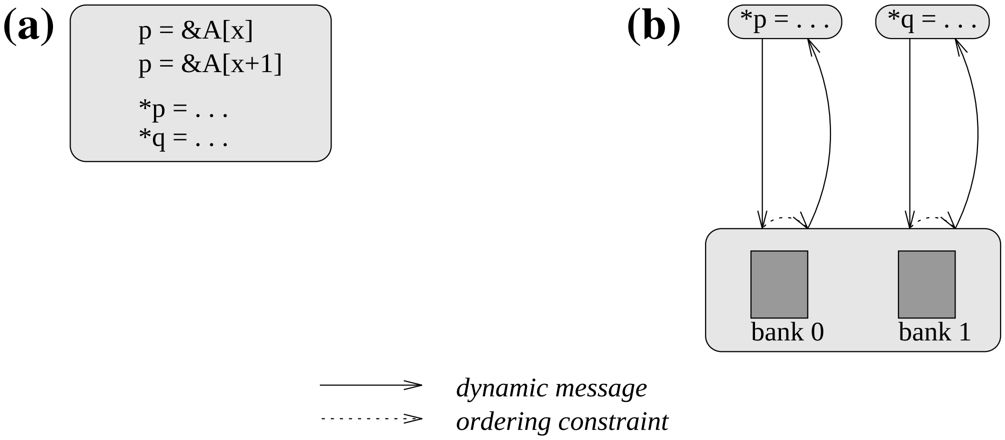


Figure 1.11: Example showing independent epochs. (a) Initial code. (b) References implemented as an independent epoch. Dependence analysis reveals that the $*p$ and the $*q$ reference are independent, and hence the references are issued as an independent epoch, without serialization.

1.5 Compiler flow

This section shows one possible compiler flow of a compiler for a software-exposed architecture. While other organizations are possible, this is the structure adopted by our compiler for the MIT Raw machine, Rawcc.

Figure 1.12 shows one possible structure for a compiler targeting a software-exposed architecture. The compiler accepts sequential programs, and automatically extracts parallelism from them for a software-exposed architecture. Rawcc has two main parts. The first part performs tasks related to the management of an exposed memory system. The Maps memory system presented in this thesis comprises this first part. The second part of the compiler is the space-time scheduler [25]. The space-time scheduler performs instruction scheduling and partitioning, as well as routing of static messages. Both Maps and the space-time scheduler are discussed in this section. Taken together, Maps and the space-time scheduler provide a unified view of distributed resources to the user, without requiring a unified view from the hardware.

The Maps memory system begins with pointer analysis and array-specific analysis, and then it performs bank disambiguation. Bank disambiguation first defines a layout of all the data objects in the program, and then tries to specify a fixed bank number, known at compile-time, for every load/store memory reference. Accesses for which a fixed bank number is found at compile-time are annotated with that bank number, and are called *disambiguated accesses*. Disambiguated accesses translate to fast, local accesses on a bank-exposed machine. Since their results are routed on the static network for Raw, disambiguated accesses are also called *static accesses*. The remaining accesses to unknown banks are called *non-disambiguated accesses* and complete over the fall-back

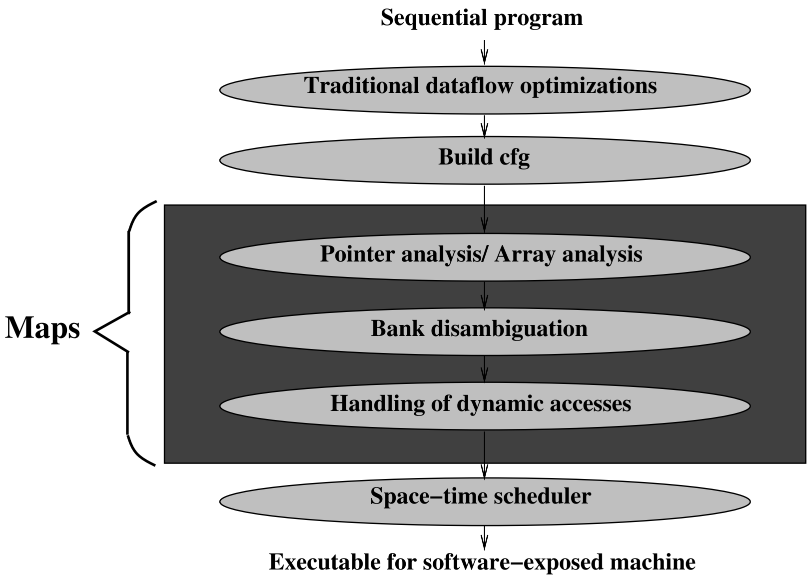


Figure 1.12: Structure of the Raw compiler.

arbitration logic. In the case of Raw, the arbitration logic is replaced by the dynamic network. Hence, non-disambiguated accesses are also called *dynamic accesses*. While disambiguation can be attained at the cost of memory parallelism by mapping all the data to one bank, our techniques provide bank disambiguation while distributing data to several banks.

Following bank disambiguation, Maps ensures that non-disambiguated (dynamic) accesses are handled efficiently. The greatest challenge to improving performance of dynamic accesses is overlapping their long access latencies with other computation and communication while respecting all dependences. This thesis presents methods to aggressively schedule dynamic accesses while respecting dependences. Methods for non-disambiguated accesses include software-serial ordering, independent epochs, and updates.

The second part of the compiler, the space-time scheduler [25], follows the analysis and code transformations in Maps. The space-time scheduler parallelizes the computation in each *scheduling unit* across the processing elements. A scheduling unit is the code granularity the space-time scheduler considers at one time. Scheduling units are basic blocks or larger; control-localization [25] extends scheduling units to regions of code with forward-control-flow only. Space-time scheduling uses the data distribution and disambiguation information provided by Maps, respecting any dependence and serialization requirements of Maps. Parallelization is achieved by statically distributing the instructions across the tiles and orchestrating any necessary communication at the register level over the static network. The decision of how to map instructions considers the tradeoffs between locality, parallelism and communication cost. Individual instruction streams proceed in a loosely synchronous manner, communicating only when there are register dependences and at the end of the scheduling units. For more details on the space-time scheduler, please refer to [25].

This compiler structure has been implemented in Rawcc, the Raw compiler built on top of SUIF [24]. Rawcc accepts a sequential program written in C or FORTRAN, and produces a Raw executable. Results in this thesis were obtained by evaluating the Rawcc compiler.

1.6 Overview of the thesis

This section presents a brief overview of the thesis. First, the scope of our compiler methods, in terms of the architectures to which they apply, is presented. Second, the contributions of this thesis are summarized as a list. Third, the organization of the thesis is presented.

Scope of our methods

The methods in this thesis apply to any bank-exposed architecture, not just those for general-purpose computing. Experimental results focus on general-purpose machines. Results are presented on the MIT Raw machine [1]. The compiler methods in this thesis apply to ILP-exploiting bank-exposed designs proposed in the past, such as the ELI-512 [16] and Iwarp [17]. The methods in this thesis also apply to many embedded system chips. Embedded chips are used for many consumer products today, ranging from cellular phones to automobiles. Embedded chips already exceed general-purpose processors in dollar volume. Many digital-signal processing (DSP) chips use more than one software-addressable memory bank [27], such as the Motorola DSP56000 family [18]. Such DSP chips with multiple software-exposed banks are bank-exposed architectures and fall within the target domain of Maps. The lack of effective compiler technology up to this point means that even today, many of these chips are hand-coded in assembly language. An exciting future application for the methods in this thesis is implementing them for DSP chips, thus aiding automatic compilation.

Contributions

This thesis makes the following contributions:

- A case is made for why software-exposed architectures exploit memory parallelism in microprocessors and why they are suited for emerging VLSI trends.

- Exploitation of memory parallelism on bank-exposed architectures is formulated as a bank disambiguation problem.

- Two techniques for bank disambiguation are developed: equivalence-class unification (ECU) and modulo unrolling.

- For the first time, pointer analysis is incorporated into bank disambiguation. Pointer analysis is used in the ECU technique.
- The modulo unrolling method is proposed for disambiguating affine-function array accesses. The method works even in the presence of non-affine accesses in the same loops as the affine functions.
- An efficient method for overlapping latencies of dynamic accesses, called software serial ordering, is proposed. Two optimizations applicable in certain cases, namely independent epochs and updates, are presented.
- The Maps system has been implemented for the Raw machine and forms part of the Raw project's common infrastructure. The Maps implementation has been used by at least 6 members of the Raw group.
- Results using Maps are presented. Runtimes are shown to decrease by factors of 3 to 5 using Maps with ILP, compared to using ILP alone.

Organization

The remainder of this thesis is organized as follows. Chapter 2 presents software-exposure as a metric for classifying architectures. Architectures that expose their resources to a higher degree than conventional microprocessors are broadly called software-exposed architectures. Three different resources – instruction-issue slots, memory banks, and the on-chip communication mechanism – are discussed, as well as the advantages and disadvantages of exposing them to the software. Finally, the MIT Raw machine, an example of the software-exposed class, is described along with its memory mechanisms.

Chapters 3 and 4 present our two methods for bank disambiguation. Chapter 3 describes equivalence-class unification, the first method for bank disambiguation. Since equivalence-class unification uses pointer analysis, the chapter first describes pointer analysis. Chapter 4 describes modulo unrolling, the second method for bank disambiguation. The chapter starts by motivating modulo unrolling through a detailed example. Modulo unrolling uses some involved mathematics in its derivation, but its use is simple. The chapter states how modulo unrolling is applied along with the formulas needed; then it goes on to prove the formulas. Finally, additional transformations needed, and optimizations possible, are described.

Our bank disambiguation schemes may choose not to disambiguate all memory references; Chapter 5 explains how non-disambiguated references can be efficiently supported. First, the chapter explains why it is difficult to efficiently handle dynamic references. Next, software serial ordering, a method for efficiently handling non-disambiguated references, is presented. Finally, two optimizations for software serial ordering – independent epochs and updates – are discussed.

Chapters 6, 7, and 8 describe implementation issues for Maps. Chapter 6 describes the compiler structure in detail. Worked examples help explain each of the 23 compiler tasks presented. Chapter 7 discusses issues relating to memory allocation and address management on bank-exposed architectures. Chapter 8 describes the implementation of certain language-specific and environment-specific features.

Chapter 9 presents results. A suite of applications compiled through Maps, containing dense-matrix, multimedia, and irregular applications, is evaluated using a simulator for the MIT Raw machine. First, numbers with and without bank disambiguation are compared. The results demonstrate that using bank disambiguation improves performance by a factor of 3 to 5 over not using it. Second, more detailed application statistics are presented, along with discussions. The final set of results demonstrate that the selective use of non-disambiguated references can improve performance in certain cases, even though non-disambiguated references have longer latencies than disambiguated accesses.

Chapter 10 discusses related work, and chapter 11 concludes. Related work in chapter 10 compares the techniques in Maps with compiler techniques discovered elsewhere, both for bank disambiguated architectures and other architectures. Other architectures are described only to an extent needed to understand their compiler techniques. For direct comparison of software-exposed architectures to other architectures, see [1].

Chapter 2

Software-exposed architectures

Software-exposed architectures are motivated by an increasing disconnect between the software view of microprocessors and their hardware implementation, and the inefficiencies arising from this disconnect. The instruction set architectures (ISA) of modern superscalars maintain a simple abstraction in software: that of a single thread of control accessing a single memory bank. In early microprocessors, such as the MIPS R2000 [28], this ISA abstraction actually reflects the hardware internals of the machine: most early microprocessors used one processing element connected to a single memory bank. Today, however, the hardware internals of modern microprocessors are more complex, using many distributed resources. For example, the AMD K7 x86 chip [29] uses six processing elements and two memory banks. To maintain the simple ISA abstraction, today's hardware designers must go to great lengths at considerable cost in area, delay and power.

Exposure of resources to the software offers one way to improve the performance of microprocessors. A resource is said to be visible to software if the machine code programming interface offers a direct way to access the resource and control its functionality. Different resources may or may not be exposed in any particular design; hence *software-exposure is a metric for classifying architectures.* Architectures that expose more resources than on conventional microprocessors are broadly called *software-exposed architectures.* Even with exposed resources, however, a unified view of the machine is desirable for the programmer; consequently in software-exposed architectures, it is the compiler's task to provide a sequential view of the machine on top of the exposed hardware.

Software-exposure in architectures is not a new idea; architectures in the past have exposed their resources to various degrees. Superscalars expose their registers to the instruction-set architecture (ISA) interface used by software. The Stanford MIPS ISA [30] exposes it its pipeline to the software; for example, through branch-delay slots. VLIWs expose their ILP available to the software, by exposing multiple instruction-issue slots to the software. Clustered VLIWs, such as the Multiflow trace [31] and the likely implementations [11] of the Intel/HP IA-64 microprocessors, expose ILP and multiple register-files to the software. Certain DSP chips, such as the Motorola DSP56000 family [18], expose their ILP, register files, and memory banks to the software, as does the ELI-512

VLIW [16] general-purpose architecture. Finally, the MIT Raw microprocessor, discussed in section 2.4, and the Iwarp [17] systolic array expose their ILP, register files, memory banks, and communication network to the software.

The downside of resource-exposure is that, to present a sequential programming interface to the user, the compiler becomes more complex. For every additional distributed resource, the compiler must orchestrate the parallelism available in that resource automatically and efficiently. The lack of suitable compiler technology has been the biggest obstacle hindering the use of exposed designs. Pressures to simplify hardware and the increasing sophistication of compiler technology have, however, contributed to a renewed interest in VLIW designs in the last 3 years [11, 13, 14, 15]. This thesis aims to further bridge the gap between compiler requirements and available compiler technology by presenting compiler methods that orchestrate multiple memory banks automatically with a high degree of memory parallelism.

The rest of this chapter is organized as follows. The first three sections, Sections 2.1, 2.2 and 2.3, discuss the exposure of three different resources, namely ILP, memory and communication, respectively. Section 2.4 describes the MIT Raw machine, which is one example of a software-exposed architecture. Section 2.5 summarizes the chapter.

2.1 Exposing ILP

This section illustrates how ILP is exposed to the low-level software by making multiple instruction-issue slots visible to the software. The interface between the fetch, decode, and execute stages of RISC pipelines for both exposed and non-exposed ILP machines is described. Architecture classes with and without exposed ILP are listed.

Exposing the instruction-level parallelism (ILP) available on chip to the software is one way designers have aimed to reduce design complexity and improve performance. Exposing ILP implies that the instruction stream interface consists of multiple instruction issue slots, rather than a single issue slot. Figure 2.1 shows machines without and with ILP-exposure. Figure 2.1(a) shows a machine without exposed ILP. ILP is hardware-discovered from a single-issue instruction stream by the decode stage, as is typical in modern superscalar pipelines [32]. Figure 2.1(b) shows a machine with exposed ILP. ILP is exposed to the software in the multiple, parallel, issue slots. Each slot of the multiple-issue instruction stream directly interfaces to its own associated functional unit, without the need for a dispatch at the decode stage. The task of discovering ILP and scheduling instructions to parallel issue slots is left to the compiler. The motivation for exposed ILP designs is that they avoid ILP-discovering hardware, along with the cost in area, power and delay of such hardware.

Both exposing ILP, and not exposing ILP, have been explored by machines in the past. The most common class of microprocessors without exposed ILP are superscalars; the most common class with exposed ILP are VLIWs. Both ILP interfaces can be used with with or without bank-exposure. Bank-exposure is discussed in section 2.2.

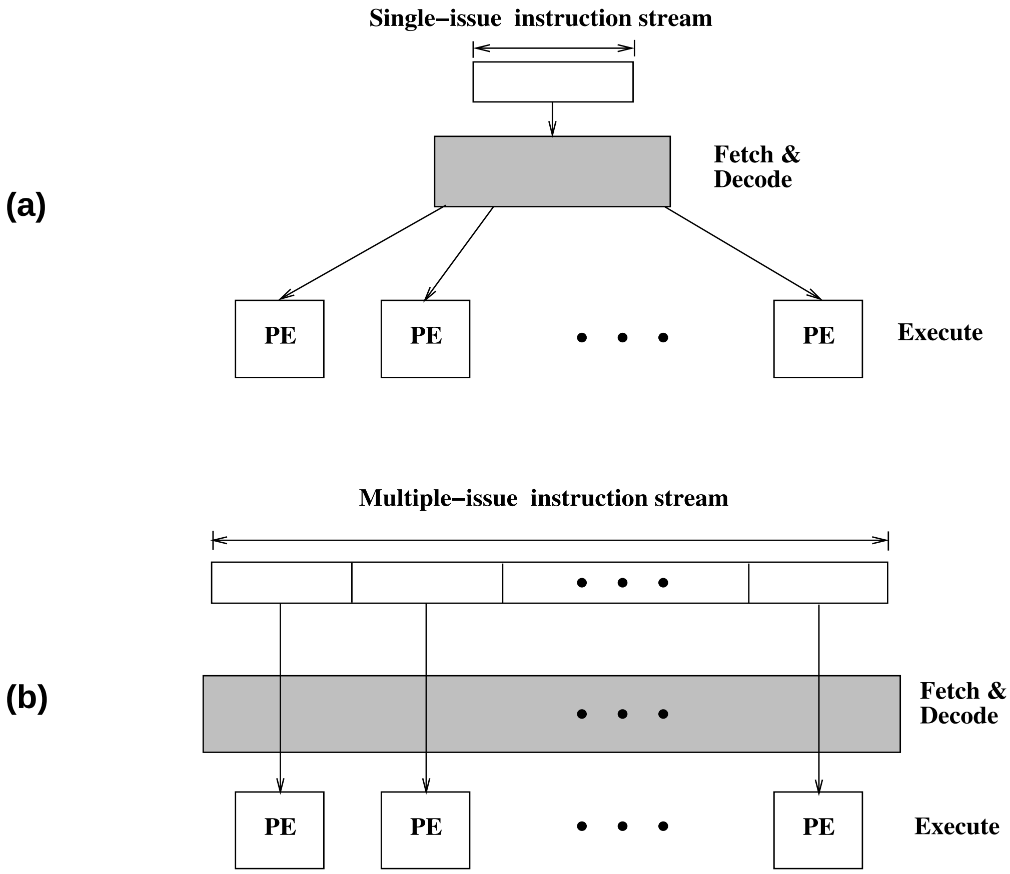


Figure 2.1: Instruction interface without and with exposed ILP. (a) Interface without exposed ILP. It shows a single instruction issue slot interfacing with a modern pipeline. ILP is discovered by the decode stage, and instructions are dispatched to multiple processing elements (PEs). (b) Interface when ILP is exposed through multiple instruction issue slots. Each issue slot interfaces with its own processing element. The compiler discovers ILP and schedules instructions into the different issue slots.

2.2 Exposing the memory system

This section shows how multiple memory banks can be exposed to the low-level software, along with the pros and cons of exposed memory. Since ILP and memory banks can be exposed orthogonally, two different kinds of bank-exposed machines result: those with and without exposed ILP. Both kinds are discussed, and a classification of architectures based on orthogonally varying ILP and memory bank exposure is presented.

Exposing the memory banks to the software is the central opportunity exploited by this thesis. A bank-exposed architecture is one that has several disjoint address spaces, corresponding to different primary cache memory banks, all visible to the software, and memory references which are directed at compile-time to particular address spaces. Compile-time resolution of the bank number is the distinguishing characteristic of bank-exposed machines: run-time resolution implies a conventional hardware-based unified memory system.

Figure 2.2 show a memory system without and with bank-exposure. Figure 2.2(a) shows a hardware-based unified memory system, implying that the banks are not exposed. Such a memory system is used in all conventional microprocessors. Other processing elements (PEs), not connected to memory, may be present. Figure 2.2(b) shows the memory system of a bank-exposed architecture.

Exposing the primary cache banks to software has two main advantages over the hardware-managed memory systems of conventional microprocessors, provided most accesses are to known banks. The first advantage of accesses to known banks is that they avoid the cost of arbitration logic used in unified memory hardware. Memory reference instructions to known banks and are either local, or use the communication network. Memory instructions to unknown banks use the arbitration logic (not shown). In conventional memory systems, in contrast, every access must go through the arbitration logic. The second advantage of accesses to known banks is that they result in on-chip locality, minimizing the penalty from wire delay. Knowing the bank accessed enables the compiler to minimize the distance from the bank to the PE where the memory reference instruction is scheduled. In the best case, the access is made local. Accesses to nearby banks will be increasingly profitable over time as wire delays across chip in terms of number of cycles are expected to rapidly increase over the coming decade. Conventional memory systems, in contrast, suffer from wire delay for every cache hit, as the wires to and from arbitration logic span large distances on the chip.

Taken together, the difficulties in scaling hardware-based unified memory systems implies that they are restricted to a small number of banks, usually 1 or 2. In contrast, bank-exposed architectures scale to dozens of banks without increase in cache latency, while increasing memory parallelism available. For example, results in chapter 9 show that on the Raw architecture, application performance continues to improve with more banks, to up to 32 memory banks.

Yet another, more indirect, benefit of bank-exposed architectures is that they employ a larger total L1 cache than conventional architectures, without increase in L1-cache latency. Today, most chip area becoming available is being dedicated predominantly to

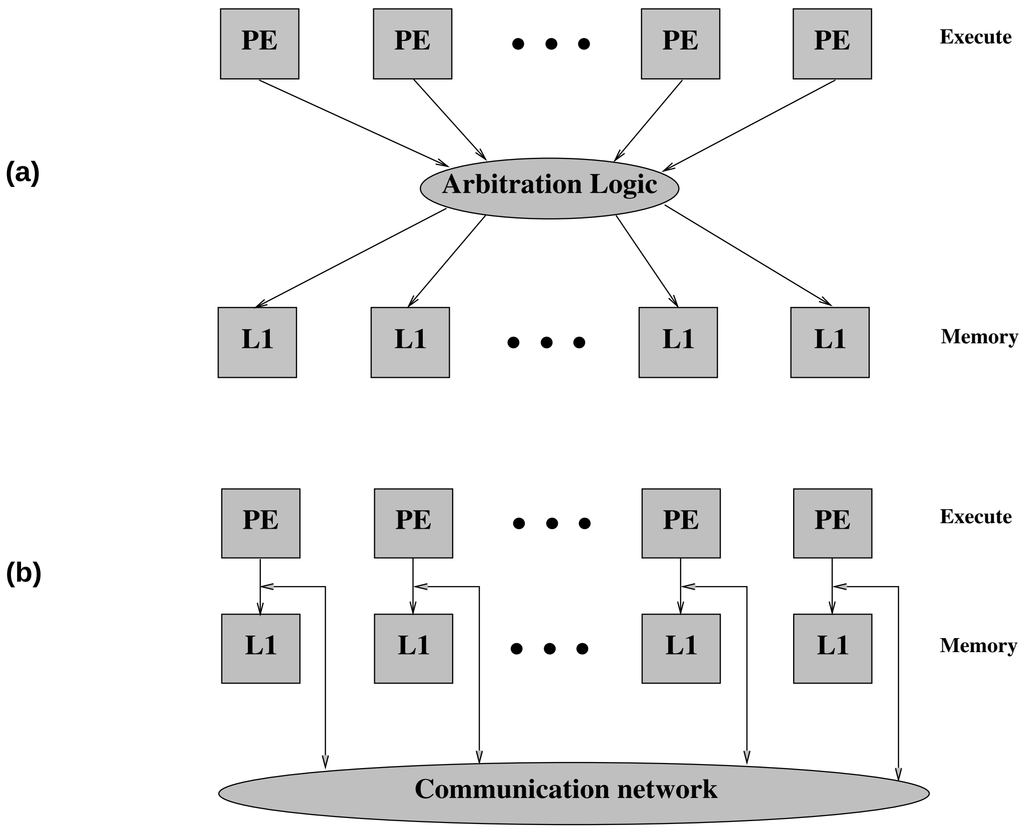


Figure 2.2: Memory system without and with bank-exposure. (a) A hardware-based unified memory system, *i.e.*, without bank-exposure. The delay through the arbitration logic and long wires are encountered by every memory access. Most microprocessors limit the number of cache banks (L1) and memory-connected PEs to a very small number (one or two) to limit the delay through the arbitration logic. (b) The memory system of a bank-exposed architecture. Accesses to banks known at compile-time complete over the communication network. Only accesses to banks unknown at compile-time need to go through the arbitration logic (not shown).

on-chip L2 cache, rather than L1. The AMD K6-III [33] has a 64K L1 cache and a 256K L2 cache, both on-chip, and the Alpha 21364 [5], to be shipped in 2000, will have a 64K L1 cache and a 1.5M L2 cache, also both on-chip. Most on-chip area is going to L2 cache rather than L1 because a cache cannot be made both large and fast. On the other hand, bank-exposed architectures have many independent L1 cache banks. Using many independent banks, exposed-memory systems have a much greater total L1-cache area than hardware-based unified memory systems with their limited number of L1 banks, without increase in cache latency.

Arbitration logic is needed for all bank-exposed architectures, but is used only for accesses to compile-time unknown banks. If the arbitration logic is exposed to the software, as in Raw, it is called a *dynamic network*. Bank disambiguated accesses use the on-chip communication network; non-disambiguated accesses use the arbitration logic.

For bank-exposed architectures, as in figure 1.5, the term *tile* is naturally defined as a memory bank with all its associated functional units. In the case of the MIT Raw machine, a tile consists of one processing element and one associated memory bank.

Exposed memory systems are relatively unexplored for microprocessors[1]. Most traditional microprocessors, including superscalars and VLIWs, use unified memory systems instead. Until now, it has not been considered feasible to guarantee at compile-time that most source-code memory instructions go to the same bank in every dynamic instance, as required for good performance on bank-exposed architectures. This thesis shows how this guarantee can be provided by the compiler, making bank-exposed architectures feasible.

Two different types of bank-exposed architectures

Exposing ILP is orthogonal to exposing memory banks, leading to two different kinds of bank-exposed machines: those that do not expose ILP, and those that do. Although both kinds of bank-exposed machines can equally use the compiler methods in this thesis, the two kinds differ. Figure 2.3 shows the two kinds of bank-exposed architecture. Figure 2.3(a) shows a bank-exposed machine without exposed ILP, obtained by combining figure 2.1(a) with figure 2.2(b). Figure 2.3(b) shows a bank-exposed machine with exposed ILP, obtained by combining figure 2.1(b) with figure 2.2(b).

The second kind of bank-exposed design, as in figure 2.3(b), has been adopted by the MIT Raw machine. The Raw machine exposes both ILP and memory. The Raw machine goes further in that it exposes its communication network as well. Other ILP-exploiting machines exposing both ILP and memory include Iwarp [17] and some DSP chips [27].

As far as we are aware, there are no machines that have adopted the first kind of bank-exposed design shown in figure 2.3(a). Bank disambiguation, however, opens up the possibility of bank-exposed machines either with or without exposed ILP. Bank-exposed machines without exposed ILP, as in figure 2.3(a), have a single-issue instruction stream, but with an extra field for load/store instructions to encode their memory bank number. The pipeline decode stage dispatches each load/store instruction to the PE/memory-bank

[1] See related works (chapter 10) for previous work in this area.

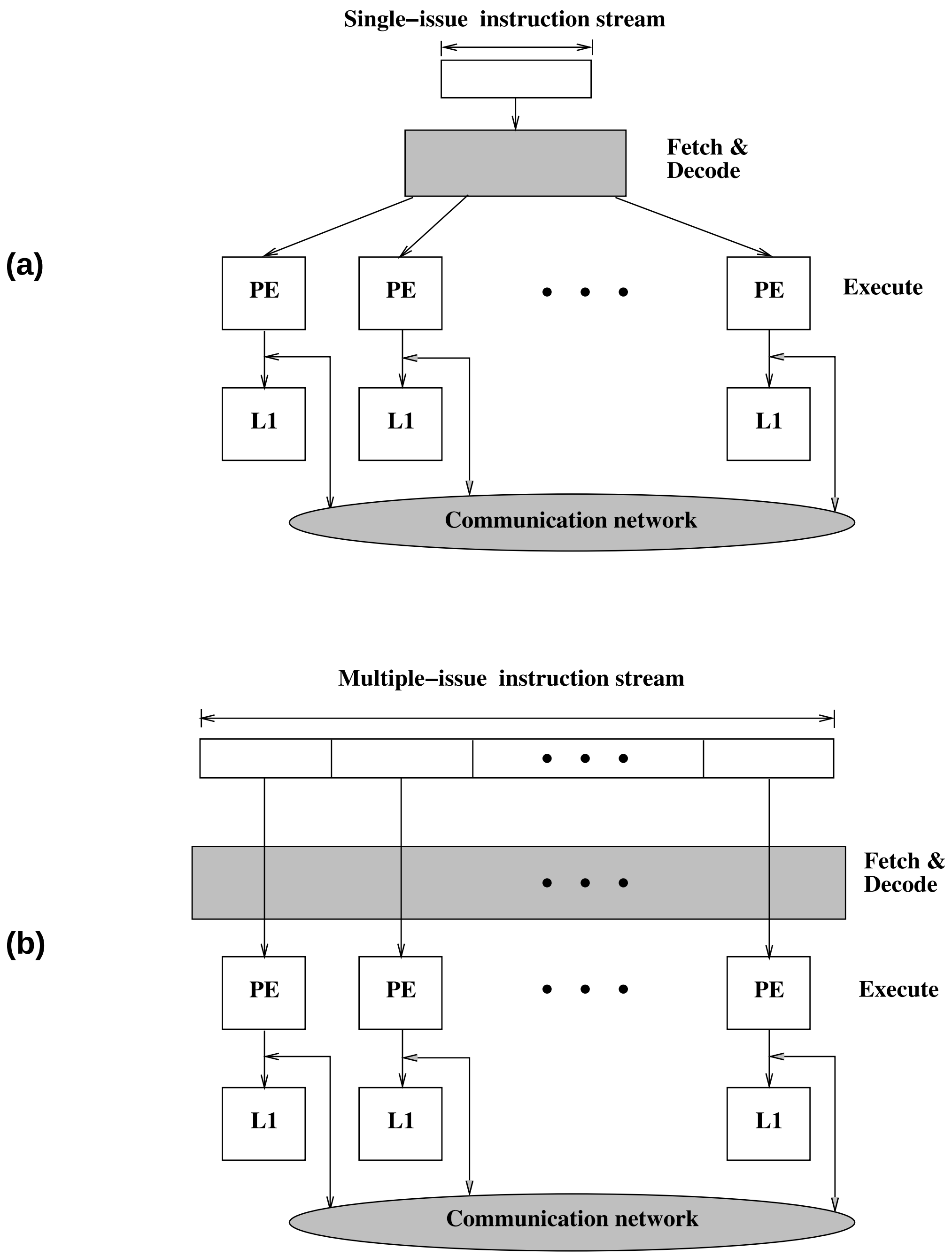


Figure 2.3: Two kinds of bank-exposed machines. (a) A machine without exposed ILP. (b) A machine with exposed ILP. The latter machine gains the full benefits of ILP exposure, but the former is an interesting partial solution. It retains a single instruction stream but with exposed banks.

pair it is assigned to. Although bank-exposed machines without exposed ILP exploit the benefits of bank-exposure, their decode-stage dispatch logic is centralized, limiting their scalability. This thesis does not study such machines further. Nevertheless, our compiler techniques apply to such machines, opening up the possibility of small microprocessors with a few exposed banks, programmed by a single instruction stream.

Classification of architectures

Since ILP and memory exposure are orthogonal, a new classification of architectures emerges. Figure 2.4 illustrates this classification. Varying ILP and memory exposure independently leads to the four possibilities shown. Within each square are salient architecture classes fitting that model. The areas shaded darker represent bank-exposed machines. The possible single-instruction-stream, bank-exposed machines refer to the as-yet-unexplored possibility discussed earlier in this section.

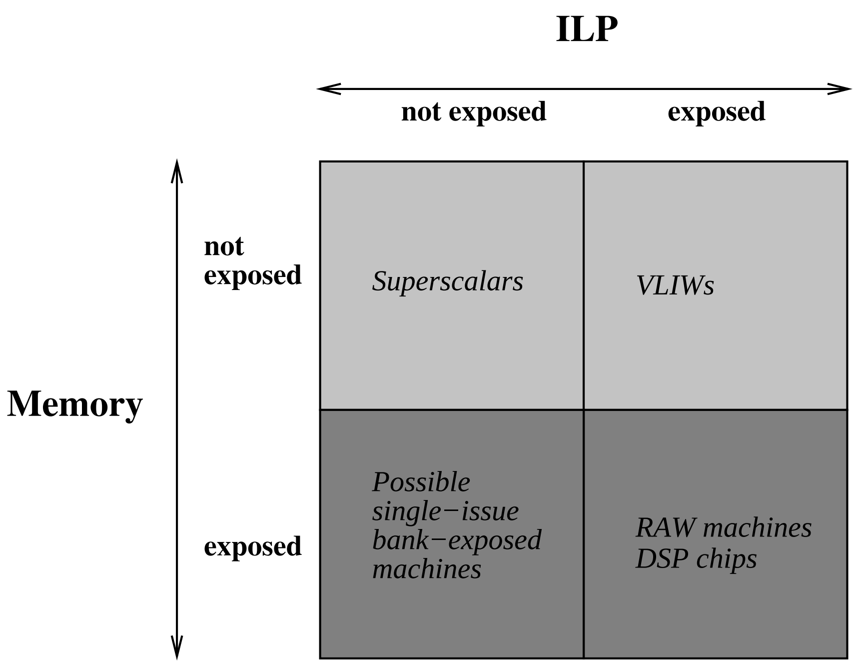


Figure 2.4: Classification of architectures. ILP and Memory exposure may be varied independently, leading to four possibilities. Each contains architectures exhibiting its characteristics. Areas shaded darker represent bank-exposed machines.

2.3 Exposing communication

This section discusses how on-chip communication can be exposed to the low-level software. All microprocessors need on-chip wires and communication control to transfer data between the physical registers and functional units. In most microprocessors this communication is *hardware-discovered*; alternatively the communication may be *software-specified*. The advantages of both are presented. Communication-exposure may expose more that message specification; software-specified communication may in addition expose intermediate-node routing, leading to *compiler-routed* messages.

All microprocessors require on-chip wires to communicate values between different physical registers, ALUs and memory banks. Exposing communication implies that on-chip communication messages are explicitly specified in the instruction steam. Such *software-specified* messages, as opposed to *hardware-generated* messages, are the hallmark of exposed communication. Most microprocessors use non-exposed schemes, such as Scoreboarding or Tomasulo's algorithm [32], that discover needed communication patterns based on program dependences automatically using hardware. For example, in Scoreboarding, a centralized hardware structure called a *scoreboard* is used. Every instruction after decoding is registered in the scoreboard along with its source and destination operands; the scoreboard keeps track of the available operands for all instructions that are awaiting issue, as well as the functional unit where issued instructions are assigned. When the operands become available, the scoreboard directs communication between the physical register where the latest value is available, to where it is used, besides updating the register file. Instructions are removed from the scoreboard after they exit the pipeline.

The advantage of non-exposed (hardware-generated) schemes for on-chip communication is compiler simplicity: the compiler need not schedule messages, as they are automatically deduced from data dependence. The disadvantage of non-exposed communication is that some form of centralizing hardware, like a scoreboard, is needed to keep track of all communication. Centralizing hardware inhibits scalability, and necessitates long wires to and from far corners of the chip. Long wires will suffer from rapidly increasing wire delays over the coming decade. Software-specified communication avoids the centralizing hardware and its disadvantages. The Raw machine uses exposed communication for both its networks to leverage the scalability and wire-delay advantages of exposed communication.

Communication may be exposed to the software beyond software-specified messages: the intermediate-node routing may also be exposed. Exposing intermediate-node routing to the software implies a *compiler-routed* network instead of a *runtime-routed network*. A message can be compiler-routed if its destination tile is known at compile-time, otherwise it must be runtime-routed. The advantage of a compiler-routed network over a run-time routed network is that the compiler orders the messages according to a static (compile-time decided) schedule, aiding static scheduling of memory and non-memory instructions [25]. The Raw microprocessor, discussed in section 2.4, provides both a compiler-routed and a runtime-routed network.

2.4 Raw architecture

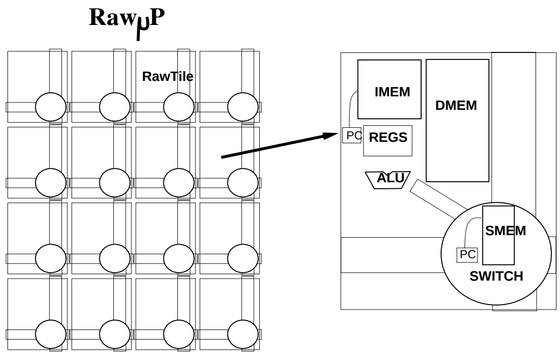


Figure 2.5: A Raw microprocessor is a mesh of tiles, each with a processing element, some memory and a switch. The processing element contains both registers and an ALU. The processing element interfaces with its local instruction memory, local data memory and local switch. The switch contains its own instruction memory.

This section describes the Raw microprocessor, a particular software-exposed architecture currently being designed in our research group at MIT. The features of its two networks are explained, along with the reason for two networks. The memory mechanisms in Raw are presented, along with their experimentally measured run-time costs.

The Raw microprocessor [1], an instance of a software-exposed architecture, exposes its ILP, memory and communication networks. Developed at our research group at MIT, the Raw microprocessor is designed to address the issue of building a scalable architecture in the face of increasing transistor budgets and wire delays that do not scale with technology. Figure 2.5 depicts the layout of a Raw machine. A Raw machine consists of simple, replicated tiles arranged in a two-dimensional mesh. Each tile, the basic unit of bank-exposed architectures, is a single processor-cache bank pair. Each tile is equipped with a switch to communicate with other tiles. The processor is a simple RISC pipeline, and the switch is integrated directly into the processor pipeline to support fast register-level communication between neighboring tiles: a word of data travels across one tile in one clock cycle. Scalability on Raw is achieved through the following design guidelines: limiting the length of the wires to the length of one tile; stripping the machine of complex hardware components; and organizing all resources in a distributed, decentralized manner.

Communication

Communication on the Raw machine is handled by two distinct networks: first, a fast compiler-routed register-level network called the *static network*, and second, a traditional runtime-routed memory-level network called the *dynamic network*. The interfaces to both networks are fully exposed to the software. Both networks consist of switches on each tile that perform message-routing.

The static network is used for messages whose source and destination tiles are known at compile-time. For such a message, the routing of the message is encoded into the instruction stream of each tile. Bank-disambiguated memory instructions can use the static network since the processing element on which the instruction is placed is decided at compile-time, and the identity of the bank accessed is compile-time-known. The static network provides *statically ordered* communication, *i.e.*, the relative order of messages arriving at each port on each tile is compiler-specified, and enforced at run-time. Message ordering is enforced by each intermediate node by routing messages in the order specified by the switch instruction stream. Statically ordered communication enables static scheduling of instructions; dependences between instructions that are mapped to different tiles are implemented by a message between them. Routing instructions in the switch instruction stream have blocking semantics that provide near-neighbor flow control; a processor or switch stalls if it is executing an instruction that attempts to access an empty input port or a full output port. Blocking semantics ensure correctness in the presence of timing variations introduced by dynamic events such as interrupts and I/O. Blocking semantics also obviate the lock-step synchronization of program counters required by many statically scheduled machines.

The dynamic network is used for messages for whom either the source or destination tile is unknown at compile-time. Such messages arise, for example, whenever a memory instruction accesses a bank whose identity is unknown at compile-time. Lacking compile-time routing information, messages on the dynamic network are run-time routed. The dynamic switch, present on each tile, is a traditional worm-hole router that makes routing decisions at run-time based on the header of each message. Processors handle incoming dynamic messages using either polling or interrupts.

The static network is faster than the dynamic network; much of the overhead in dynamic networks is because of the unpredictable timing and routing of dynamic messages. Three reasons for why the dynamic network is slower are as follows. First, because of unpredictability in the arrival order and timing of messages, expensive reception mechanisms such as polling or interrupts are required. On the static network, the exact ordering of messages is known; hence a processor expecting a static message blocks until the message arrives, without needing polling or interrupts. Blocking time is low, and often zero, as timing can be predicted on the static network barring dynamic events such as interrupts and I/O. Processors block for static messages only at the times they expect incoming messages. A second reason why the dynamic network is slower is that no overhead words are needed in a static message other than the data; dynamic messages must include a destination tile, and a handler address or message ID. Overhead words in messages require time to compose, launch, route and handle. A third reason why the dynamic network is slower is that it may incur congestion costs; when more messages arrive at a tile than can fit in the incoming message queue, some messages are spilled to memory. Spilling to memory is an expensive operation; in contrast, spilling is never needed on the static network, as a message does not arrive on a tile until its preceding message, if any, is handled.

Why have two networks?

Having two networks for inter-tile communication goes against the principle of unity of mechanism; why provide two networks when one would suffice? Indeed, there are disadvantages in providing two networks. Two networks take up more chip area than one, and require two interfaces from the processor pipeline instead of one. Nevertheless, there are arguments supporting the use of two networks. Results in chapter 9 show that using both networks provides improved performance over using the dynamic network alone, often by a factor of 5 or more. This result argues that the static network is valuable. Even when the dynamic accesses are optimized using independent epochs and updates (section 5.6), performance is competitive with the static-only case only for 32 or more tiles. Moreover, independent epoch and update optimizations are not applicable for all dynamic accesses.

We have argued for the static network; however, it is less clear why the dynamic network is needed. The methods in this thesis can eliminate all non-disambiguated dynamic accesses if desired. The dynamic network is included in Raw not so much because of the recommendations of this thesis, but to allow for greater functionality in connections with off-chip devices such as DRAM memory and I/O devices. Scheduling of unpredictable events, such as cache misses, interrupts and I/O messages, is not possible on the compiler-scheduled static network. Another, lesser, factor arguing for a dynamic network are the results in chapter 9 that show that limited use of the dynamic network improves performance is some cases.

Memory mechanisms on Raw

From its communication mechanisms, the Raw architecture provides three ways of accessing memory: local access, remote static access, and dynamic access, in increasing order of cost. A memory reference can be a local access or a remote static access if it satisfies the *static residence property* — that is, (a) every dynamic instance of the reference must refer to memory on the same tile, and (b) the tile must be known at compile time. The access is local if the Raw compiler places the subsequent use of the data on the same tile as its memory location; otherwise, the access is a remote static access. A remote static access works as follows. The processor on the tile with the data performs the load, and places the loaded value onto the output port of its static switch. Then, the pre-compiled instruction streams of the static network route the loaded value through the network to the processor needing the data. Finally, the destination processor accesses its static input port to get the loaded value.

If a memory reference fails to satisfy the static residence property, it is implemented as a dynamic access. A load access, for example, turns into a split-phase transaction requiring two dynamic messages: a load-request message followed by a load-reply message. Figure 2.6 shows the components of a dynamic load. The requesting tile extracts the resident tile and the local address from the "global" address of the dynamic load. The requesting tile sends a load-request message containing the local address to the resident tile. When a resident tile receives a request message, it is interrupted, performs the load of the requested address, and sends a load-reply with the requested data. The tile needing the data eventually receives and processes the load-reply through an interrupt, which stores the received value in a predetermined register and sets a flag. When the resident tile needs the value, the resident tile checks the flag and fetches the value when the flag is set. The request for a load need not be on the same tile as the use of the load.

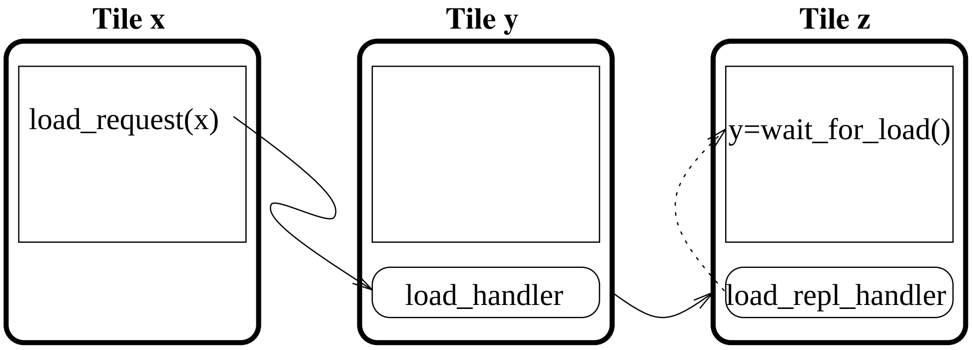


Figure 2.6: Anatomy of a dynamic load. A dynamic load is implemented with a request and a reply dynamic message. The request for a load need not be on the same tile as the use of the load.

Distance	0	1	2	3	4
Dynamic store	17	20	21	22	23
Static store	1	4	5	6	7
Dynamic load	28	34	36	38	40
Static load	3	6	7	8	9

Table 2.1: Cost of memory operations in processor cycles

Table 2.1 lists the end-to-end costs of memory operations as a function of the tile distance. The costs include both the processing costs and the network latencies. Figure 2.7 breaks down these costs for a tile distance of 2. The measurements show that a dynamic memory operation is significantly more expensive than a corresponding static memory operation. Part of the overhead comes from the protocol overhead of using a general network, but much of the overhead is fundamental to the nature of a dynamic access. For example, a dynamic load requires sending a load request to the proper memory tile, while a static load can optimize away such a request, because the memory tile is known at compile time. The need for flow control and message atomicity to avoid deadlocks further contributes to the cost of dynamic messages [34]. Finally, the inherent unpredictability in the arrival order and timing of messages requires expensive reception mechanisms such as polling or interrupts. In the static network, blocking semantics combine with compile-time ordering and scheduling of static messages to obviate the need for expensive reception mechanisms.

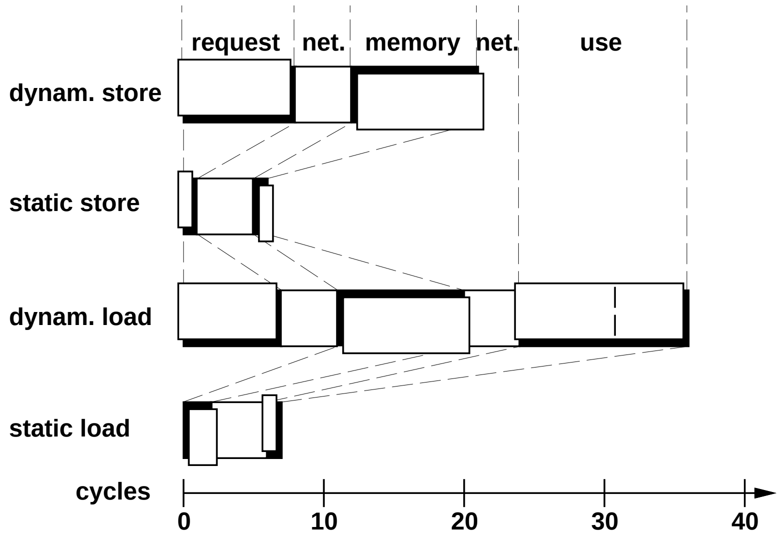


Figure 2.7: Breakdown of the cost of memory operations between tiles 2 units apart. Highlighted portions represent processor occupancy. Unlifted portions represent network latency.

2.5 Summary

A summary of this chapter follows. The chapter begins by showing that software-exposure is a metric to classify architectures. A rationale for software-exposure is presented, *i.e.*, less hardware to automatically manage resources, and consequently, savings in delay though the hardware. Subsequently, the exposure of three different resources – instruction-issue slots, memory banks, and communication wires – are discussed. A classification of architectures based upon orthogonally varying the exposure of instruction-issue slots and memory banks is presented. Finally the MIT Raw architecture is described, along with its motivations. The Raw architecture uses two networks; the advantages and disadvantages of each are mentioned. Finally, the different ways of accessing memory in Raw are described, and their costs compared.

Chapter 3

Equivalence-class unification

This chapter describes equivalence-class unification, our first method for bank disambiguation. Equivalence class unification (ECU) is used by Maps as a baseline technique, *i.e.*, all accesses by default use ECU for bank disambiguation. Arrays with mostly affine accesses are promoted using modulo unrolling, our second technique for bank disambiguation, described in chapter 4. ECU handles all kinds of accesses in an integrated framework, including non-affine array accesses, pointer dereferences, structure references and heap references. ECU handles programs with arbitrary memory-aliasing. ECU does not transform the code, hence the code size is not increased. Instead, ECU provides disambiguation by using intelligent data partitions. ECU can always bank disambiguate all references, although the amount of memory parallelism exposed may vary, depending upon the characteristics of the program.

This chapter is organized as follows. Section 3.1 describes pointer analysis, a traditional technique, leveraged by Maps and ECU in particular. Section 3.2 describes the ECU method, and shows an example of its use. Section 3.3 discusses the quality of the disambiguation ECU provides, and ways to improve it. Section 3.4 summarizes the chapter.

3.1 Pointer analysis

This section describes pointer analysis, a compiler analysis technique discovered elsewhere, that is needed to understand ECU. The information provided by pointer analysis is described, and three uses of pointer analysis in Maps are discussed: for ECU; for minimization of memory dependences; and for software serial ordering, a technique for non-disambiguated accesses.

In order to understand ECU, pointer analysis [35, 36] needs to be understood first. Pointer analysis is a compile-time technique that, for every memory-reference instruction, determines the data objects that the instruction can possibly refer. Maps uses SPAN [35], a state-of-the-art pointer-analysis package that provides an inter-procedural, flow-sensitive and context-sensitive pointer analysis.

To understand pointer analysis, consider the input program in figure 3.1(a). Figure 3.1(b) shows the results of the SPAN pointer analysis package on the code in figure 3.1(a). SPAN assigns a unique number, called a *location-set number*, to each abstract object in the program. An abstract data object is either a stack-allocated variable declaration in the program, or a group of dynamic objects created by the same heap-memory allocation call site in the program. Figure 3.1(b) shows the abstract data objects marked with *assign* comments, with the location set numbers for the objects listed alongside the comment. An entire array is considered a single object, but each field in a structure is considered a separate object. Finally, pointer analysis annotates each memory reference instruction with a *location-set list*, a list of location-set numbers corresponding to the objects that the memory instruction can refer. In figure 3.1(b), each memory instruction is annotated with its location-set list, shown as *ref* comments. For simplicity, location-set numbers are shown only for objects that have program load/stores to them; in reality all objects are assigned such numbers.

Uses of pointer analysis

Maps uses pointer analysis for three purposes: minimization of dependence edges, equivalence-class unification (ECU), and software serial ordering (SSO). First, pointer analysis is used to minimize memory-dependence edges. With no information, all memory references are potentially dependent, and hence every pair of memory instructions of which at least one instruction is a store are connected by a dependence edge. With pointer analysis, however, more precise information is available. A pair of memory instructions have a dependence edge between them if and only if the intersection of their location-set lists is non-empty[1], and at least one of the instructions is a store. Figure 3.1(b) shows this rule for inserting dependence edges: edges, shown with dotted arrows, are present between memory instructions only when the intersection of the referenced location-set lists is non-empty.

The second use of pointer analysis is in ECU, described in section 3.2. The third use for pointer analysis is in SSO: SSO uses equivalence classes to assign a turnstile per class; equivalence classes are derived from pointer analysis information in ECU. SSO is described in section 5.4.

3.2 Equivalence-class unification method

This section describes equivalence-class unification, the first of two bank disambiguation schemes presented in this thesis.

[1]For dependences between array references, we can do better. Pointer analysis does not distinguish between references to different elements in an array, so that reference pairs such as $A[1]$ and $A[2]$ are analyzed to be dependent. For array references, Maps, in addition, uses traditional array dependence analysis to obtain finer grained dependence information [37].

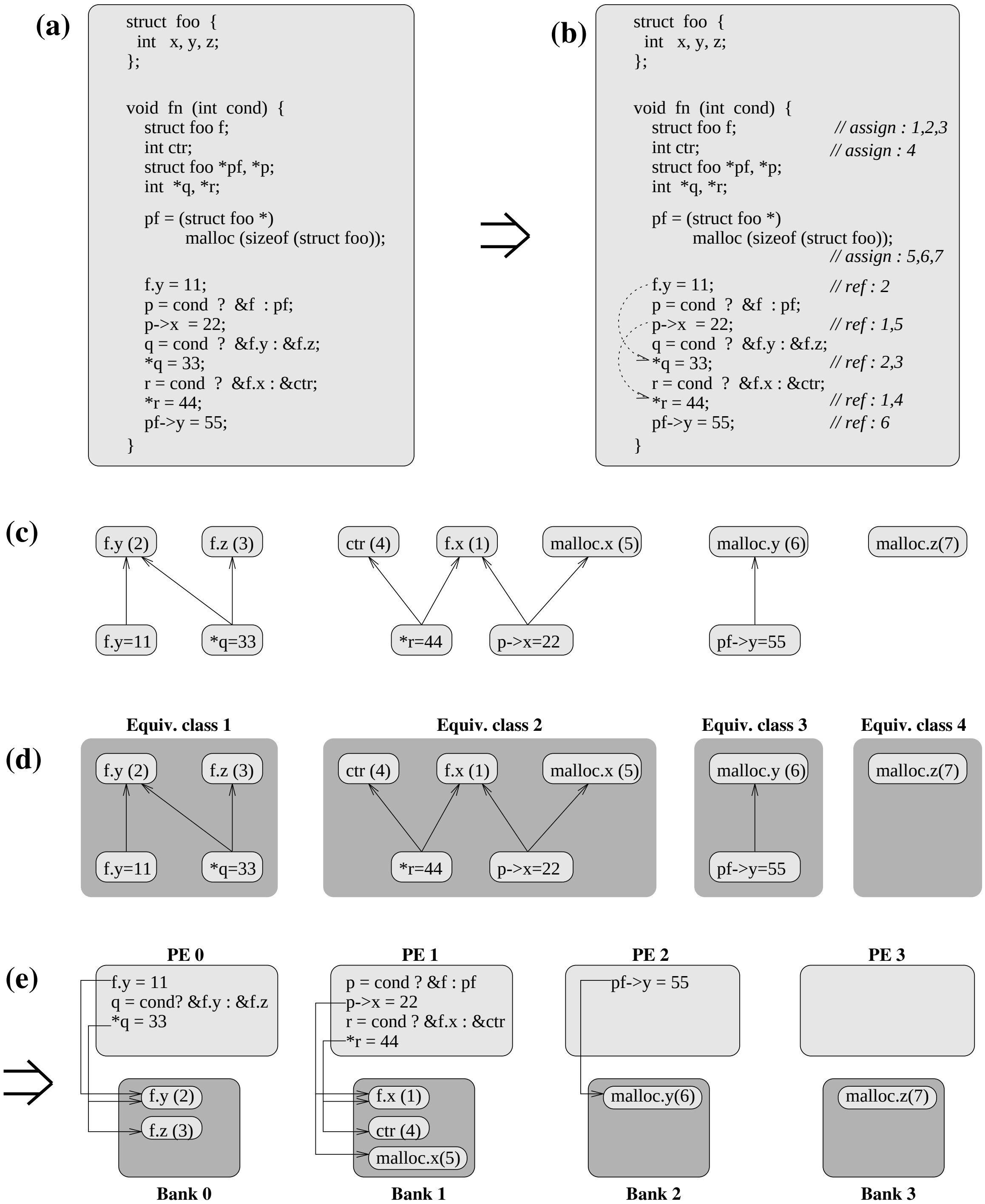


Figure 3.1: Example showing equivalence-class unification. (a) Initial program. (b) After pointer analysis, showing location-set numbers and dependence edges (dotted lines). (c) Memory objects and instructions represented as bi-partite graph. (d) Connected components of bi-partite graph marked as equivalence classes (ECs). There are 4 ECs: {2,3}, {1,4,5}, {6}, {7}. (e) After code generation. PE = Processing element. Each EC is mapped to a single bank; the code is distributed among the different PEs while ensuring that reference instructions are local to the bank they access.

Equivalence class unification (ECU) is a technique that uses intelligent data distribution to achieve bank disambiguation. Figure 3.1 helps illustrate the steps involved in ECU through an example. As a first step, ECU runs pointer analysis: figure 3.1(b) shows the results of pointer analysis on the program in figure 3.1(a). Each data allocation site is annotated with a location-set list, where each location-set corresponds to a data object allocated by that site. Each memory instruction is annotated by the location-sets corresponding to the data objects the instruction can reference. Dependence edges are also computed using pointer analysis.

Next, ECU represents the pointer analysis information as a bipartite graph of data objects and memory references. Figure 3.1(c) shows the bipartite graph for the program in figure 3.1(b). A node is constructed for each abstract object, and for each memory reference instruction. In the figure, the upper row shows the abstract objects, with the location-set number for each object in parenthesis, and the lower row shows the memory reference instructions. Edges are constructed from each memory reference to the all the abstract objects whose numbers are in the reference's location-set list.

Subsequently, ECU defines the concept of *alias equivalence classes* using the bi-partite graph. Alias equivalence classes form the finest partition of the location set numbers such that each memory access refers to location-set numbers in only one class. Figure 3.1(d) shows the bi-partite graph in figure 3.1(c) partitioned into 4 equivalence classes. Maps derives the equivalence classes by computing the connected components of the bi-partite graph for the program. The location-set numbers in each connected component form a single alias equivalence class, along with the memory-reference instructions to those location sets. Instructions in the same alias class potentially refer to the same object, while references in different classes never refer to the same object.

Finally, each equivalence class is mapped to a single tile to achieve bank disambiguation. Mapping each class to a single bank disambiguates all memory instructions, as all the data objects the instruction possibly accesses are in one connected component. Since each connected component is mapped to a single bank, all instructions are bank-disambiguated. Figure 3.1(e) shows the results of code generation using the equivalence classes in figure 3.1(d). Equivalence class 1 is mapped to bank 0, class 2 to bank 1, class 3 to bank 2, and class 4 to bank 3. This particular class-to-bank mapping is incidental; any mapping from classes to banks can be used. The advantage of mapping different equivalence classes to different banks is that memory parallelism between accesses to different classes is exploited. Since equivalence class unification does not depend on any particular application characteristic, it extracts available parallelism from all classes of programs, even those with irregular access patterns and heavy pointer usage.

Rather than specifying physical banks, ECU specifies a virtual bank number for each class, assuming an infinite number of virtual banks. If the number of physical banks is less than the number of virtual banks used, a many-to-one virtual-to-physical mapping is done. The space-time scheduler [25] performs the virtual-to-physical mapping with the aim of choosing mappings in which virtual banks that are rarely accessed concurrently are mapped to the same physical bank.

Figure 3.1(e) leaves certain implementation details unspecified, for example, how to implement distributed malloc() calls, or pointers to distributed objects. Chapter 7 describes the implementation of distributed objects; chapter 8 describes the implementation of procedure calls.

3.3 Quality of the disambiguation

This section discusses the quality, in terms of memory parallelism, of the disambiguation provided by ECU, and how it may be improved in certain cases.

The quality of the disambiguation offered by strictly applying ECU depends upon the number and size of the alias classes. A large number of small classes gives the most memory parallelism, as accesses mapped to different classes can execute in parallel. The number and size of the classes depends upon the access patterns of the program, which the compiler cannot control. The compiler can, however, improve upon ECU by applying it with modifications, or performing additional transformations. This section describes three ways in which ECU can be improved: sub-dividing aggregate objects, detecting 'bad' references and making them non-disambiguated, and finally, cloning procedures.

First, dividing single location sets further into multiple banks may allow us to improve upon ECU. Sub-dividing single location-sets may be profitable for aggregate data objects such as arrays and structures, as parallelism is often available between accesses to different parts of such objects. That ECU does not distribute arrays is the motivation for modulo unrolling, the second disambiguation scheme in this thesis. Modulo unrolling, discussed in chapter 4, exploits parallelism within arrays. Structures need not be handled separately as ECU itself finds parallelism within structures. SPAN differentiates between accesses to different fields, so that fields of a structure can be in different alias equivalence classes and distributed across the tiles.

Second, allowing some accesses to remain non-disambiguated may help improve upon ECU. Consider a case when a single 'bad' reference causes ECU to map several location sets into a single equivalence class, yet in most of the program, those location sets are accessed in parallel by other references. If profiling data can reveal this situation by comparing execution frequencies, an optimization may be possible. It may be profitable to keep the location sets in separate classes at the cost of making the infrequent 'bad' access non-disambiguated, and hence slow. Keeping the location sets in different banks enables the more common parallel accesses to go to different banks, preserving their parallelism. This optimization involving 'bad' references has not been automated; hence all the performance improvements in our results were obtained without it. Studies involving the selective use of non-disambiguated accesses are presented in section 9.3.

Finally, cloning of procedures before ECU may improve the performance of ECU. Consider a procedure having a pointer for a formal parameter, and two call sites whose actual argument values point to different location-sets. If the procedure body dereferences the pointer, then ECU maps both location sets to the same class in order to disambiguate the dereference. One way to improve performance is to clone the procedure into two copies, one for each call-site. With procedure cloning, the two location sets are not mapped to the same class, which is helpful if they are accessed in parallel elsewhere in the program.

3.4 Summary

A summary of the chapter follows. Equivalence class unification (ECU) is described; ECU is shown to be a effective method for providing bank disambiguation with memory parallelism, and without any increase in code size. Disambiguation is attained by carefully distributing the data and memory instructions; the code is not transformed. Pointer analysis, a traditional compiler technique is used to guide data placement in ECU. Pointer analysis is also used in Maps for minimizing dependence edges, and in software serial ordering, a technique for handling non-disambiguated accesses (chapter 5). This chapter concludes by showing scenarios in which ECU can be improved, and how. The lack of memory parallelism within arrays in ECU is the motivation for our second method for bank disambiguation, modulo unrolling, described in chapter 4.

Chapter 4

Modulo unrolling

This chapter presents modulo unrolling, our second bank disambiguation method. Modulo unrolling is applicable for programs having array references whose index expressions are predominantly affine functions of enclosing loop induction variables [1]. Modulo unrolling is provably able to bank disambiguate all affine accesses. Affine array accesses, along with scalar variables, form the bulk of the accesses in dense-matrix scientific codes, as well as some multimedia and streaming applications.

This chapter is organized as follows. Section 4.1 motivates modulo unrolling, and outlines it using a simple example. Section 4.2 states the modulo unrolling method in terms of a formula for the unroll factors, and presents the scope of the method. Section 4.3 derives the formulas for the minimum unroll factors required. Section 4.4 discusses issues related to the code growth in modulo unrolling. Upper bounds for the code growth are derived, and an optimization called the padding optimization is shown to reduce the code growth in certain cases. Up to this point, modulo unrolling guarantees that all targeted array references refer to memory on a single bank, but the identity of that bank may not be determinable at compile time. Section 4.5 outlines an additional transformation that is required in some cases to make this bank compile-time determinable. Section 4.6 describes affine code generation. It shows how the bank numbers and local offset expressions are actually determined. Section 4.7 discusses optimizations for array references, other than modulo unrolling. Section 4.8 summarizes the chapter.

4.1 Motivation and example

This section motivates and describes modulo unrolling through the use of a step-by-step example.

The major limitation of equivalence class unification is that an array is treated as a single object belonging to a single equivalence class. Mapping an entire array to a single

[1] An *affine function* of a set of variables is defined as a linear combination of those variables, plus a constant. As an example, given i,j as enclosing loop variables, $A[i+2j+3][2j]$ is an affine access, but $A[ij+4]$ and $A[2i^2+1]$ are not.

memory bank sequentializes accesses to that array and destroys the parallelism found in many loops. Therefore, arrays with affine accesses are not disambiguated using ECU; instead they are disambiguated using modulo unrolling.

Figure 4.1 shows an example of how bank disambiguation is done using modulo unrolling. Figure 4.1(a) shows the initial code fragment consisting of a simple **for** loop with three array accesses inside the loop. The data space depicts the view from the compiler at that point; at the input, the programmer views all data objects, $A[]$ and $B[]$ in this case, to be in one unified address space. Both arrays are assumed to range from 0 to 99.

Figure 4.1(b) shows the array distributed using low-order interleaving. In low-order interleaving, successive array elements are interleaved among the N banks in a round-robin manner, beginning at bank 0. More formally, array $A[]$ is broken up into N sub-arrays, $A_0[]$ to $A_{N-1}[]$, such that $A[i]$ maps to $A_{i \bmod N}[i \text{ div } N]$. Low-order interleaving is desirable since spatially-close array accesses, such as $A[i]$ and $A[i+1]$, are also often temporally close. Low-order interleaving places these on different banks, thus allowing for parallelism in their accesses. For certain programs, more tailored layouts may improve performance, but that would destroy the uniformity that makes bank disambiguation and static promotion workable without extensive inter-procedural analysis. Unfortunately, all 3 array references each access all 4 banks, as depicted by the arrows from each of the 3 references to all 4 banks. The $A[i]$ reference, for example, has the bank-access pattern of 0, 1, 2, 3, 0, 1, 2, 3, ... in successive loop iterations. Since it goes to more than one bank, it fails disambiguation.

A naive way to attain bank disambiguation and memory parallelism is to *fully unroll* the loop. For the code in figure 4.1(b), the loop could be unrolled by a factor of 100. Full unrolling of the loop makes all the array-reference indices constant in the unrolled code; hence all the array references are trivially disambiguated to the bank holding the known array element. Full unrolling provides disambiguation with memory parallelism; however, full unrolling is prohibitively expensive in terms of code-size increase. Full unrolling is not even possible for compile-time-unknown loop bounds. Consequently, the Maps system never uses full unrolling. The challenge is to attain disambiguation and memory parallelism in a method that keeps code growth bounded, and works for unknown loop bounds.

In this case, there is a way to attain both memory parallelism and bank disambiguation, without a huge increase in code size. The solution stems from the observation that in the original code, the bank-access pattern for all 3 accesses repeat themselves after a fixed distance of 4. For example, for the $B[i+i]$ access, the bank-access pattern of 1, 2, 3, 0, 1, 2, 3, 0 ... is a repetition of the pattern 1, 2, 3, 0. Consequently, a loop-unroll by a factor of the repetition distance, 4 in this case, has the property that in the unrolled loop, each array reference will go to a single bank. Figures 4.1(c) through 4.1(e) illustrate bank-disambiguation through modulo-unrolling. Figure 4.1(c) shows the loop unrolled by a factor of 4. Figure 4.1(d) shows that in the unrolled loop, each access goes to only one memory bank, enabling the original references to be replaced by references to the sub-arrays A_0 to A_3, and B_0 to B_3. For example, the two $A[i+1]$ references in figure 4.1(c) access data elements $A[1]$, $A[5]$, $A[9]$, $A[13]$, ...; since all these elements are

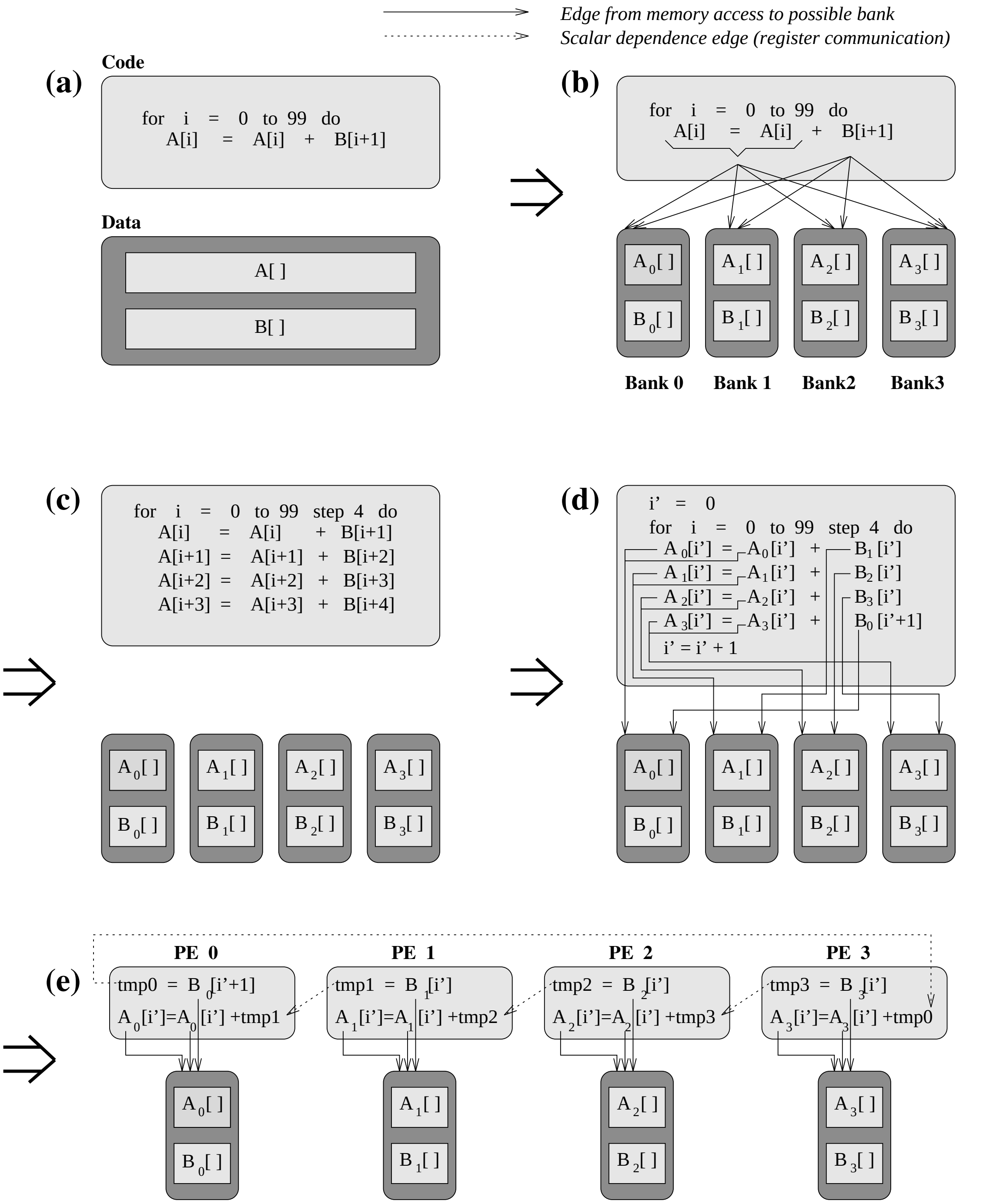


Figure 4.1: Example showing modulo unrolling. Pictures show sample code and a view of memory after successive compiler steps for a 4-banked memory. (a) Initial program and unified view of memory. (b) After low-order interleaving. (c) After modulo unrolling step 1. (d) After modulo unrolling step 2. (e) After code generation, PE=processing element. In (b), each memory instruction goes to all 4 banks in different iterations. In (d) and (e), each memory instruction goes to a fixed bank number known at compile-time.

mapped to sub-array $A_1[]$, the $A[i+1]$ references are replaced in figure 4.1(d) by references to $A_1[i']$. The index i' is a local index ranging from 0 to 24, replacing the original index i that ranged from 0 to 99; i' is incremented by 1 for every iteration of the unrolled loop, instead of 4. The transformations needed to convert the code in figure 4.1(c) to that in figure 4.1(d) are described in chapter 6.

Figure 4.1(e) shows the generated code for the 4 processing elements (PEs) associated with the 4 banks in the bank-exposed architecture. The sub-array reference instructions are all local, *i.e.*, a sub-array reference is placed on the PE local to the bank on which the sub-array resides. The placement of references in figure 4.1(e) implies that communication is required. For example, the add instruction that adds $A_0[i']$ and $B_1[i']$ is scheduled on PE 0, but since $B_1[i']$ is loaded on PE 1, the value of $B_1[i']$ is loaded into a temporary variable $tmp1$ that is shipped over to PE 0 using the register-communication network. A register-communication message is needed for every data-dependence that goes from one PE to a different PE. Register communication always involves compile-time-known messages; hence, compile-time scheduling of the messages is possible. On Raw, the static network is used to communicate register communication values.

The program transformation in figure 4.1 is an example of the general technique of *modulo unrolling* described in this chapter. Modulo unrolling is a technique that provably disambiguates any array reference in a nested loop if the indices of the array reference are affine functions of the index variables of the enclosing loops. The modulo unrolling transformation provides both bank disambiguation and memory parallelism. Section 4.3 proves that unroll factors exist for all loops such that affine functions within the loop nests are disambiguated upon unrolling, and derives a formula for the precise unroll factors required.

The price to pay for modulo unrolling is increased code size from unrolling. Fortunately, section 4.4 shows that the unroll factors required depend only upon the number of banks and the affine-function characteristics, and are independent of the loop bounds. The unroll factor for most common cases is upper-bounded by N, the number of banks, and hence, the code-growth is modest. Further, the negative impact of code growth on I-cache performance is likely to be limited for the following reason. The code is divided into N different I-caches for a machine with multiple instruction streams, such as the MIT Raw machine. For low-order interleaved arrays accessed by affine references, it is likely that all the I-caches miss simultaneously, for example, when the loop is reached the first time. Simultaneous I-cache misses are desirable as they overlap miss latencies with each other, thus reducing the performance penalty from

4.2 Modulo unrolling method

This section formally states the method for modulo unrolling, including the formula for the unroll factors used. The formula is derived later in section 4.3. The scope of nested loops to which modulo unrolling applies is described.

The method for modulo unrolling is as follows. First, the compiler looks for affine array accesses inside loop-nests. Let k be the number of loops in the loop nest for any one affine function. Next, the compiler computes the unroll factor U_j, for all j in $[1, k]$, induced by each affine array access on the jth loop from its enclosing loop-nest. U_j is computed using the following formula, which uses D_j as an intermediate variable.

$$D_j = N \ / \ \gcd\left(N, \sum_{i=1}^{d}\left(c_{i,j} \prod_{l=i+1}^{d} MAX_l\right)\right)$$

$$U_j = \text{lcm}(D_j, s_j) \ / \ s_j$$

where

k is the number of loops in the loop nest enclosing the affine function;
d is the number of array dimensions in the affine array access;
v_j is the induction variable of the jth loop in the loop nest $(1 \leq j \leq k)$;
$c_{i,j}$ is the constant coefficient of v_j in the ith array dimension of the affine access $(1 \leq i \leq d, 1 \leq j \leq k)$;
$c_{i,k+1}$ is the constant factor in the ith array dimension of the affine access $(1 \leq i \leq d)$;
N is the number of memory banks in the target software-exposed architecture;
MAX_i is the size of the ith array dimension $(1 \leq i \leq d)$;
s_j is the step size of the jth loop in the loop nest $(1 \leq j \leq k)$.

All symbols defined above, and in the rest of this chapter, are integers. The formula for U_j is derived in section 4.3. Once the compiler computes the induced unroll factors for each loop for each affine access, the final unroll factor for any loop is computed as the lcm of the unroll factors induced by all enclosing affine array accesses. Taking the lcm of the unroll factors induced by all the array references is needed, as we want to disambiguate all the references, each possibly demanding a different unroll factor. Next, the additional transformation described in section 4.5 is performed when needed. Finally, code generation occurs as described in section 4.6.

While a row-major layout is assumed to prove the result in the unroll-factor theorem, a different layout may be used for the remaining code generation. The row-major choice in the unroll-factor theorem is used to partition the array and computation in one particular manner: once that partitioning is done, any ordering can be used on the resultant new local arrays.

Scope

Modulo unrolling handles arbitrary affine functions with few other restrictions. Within its framework, it handles imperfectly nested loops, non-unit loop step sizes, hand-linearized multidimensional arrays, and unknown loop bounds. First, imperfectly nested loops are automatically handled with no penalty or special cases, as the derivation in section 4.3

did not assume perfect nesting. Second, non-unit loop step sizes are handled as they are integrated into the framework. Third, the method works unchanged even if multi-dimensional arrays are hand-linearized by the programmer. Hand-linearization preserves the affine property: a linear combination of affine functions is also affine, and the offset of any array element from its base remains unchanged using hand-linearization. Finally, modulo unrolling handles unknown loop bounds, but code with unknown lower bounds may require an additional transformation, as explained in section 4.5.

4.3 Deriving the unroll factors

This section proves that unrolling each loop in a loop nest by a certain factor disambiguates all affine array accesses in that nest. The proof also derives the formula for the unroll factor U_j that yields the minimum overall code growth. The symbols used are defined in section 4.2. Additional symbols needed in the proof are defined when needed.

A roadmap for the proof follows. First, two supporting theorems involving modular arithmetic are proved, namely, the product modulo theorem (Theorem 4.1) and the sum-of-products modulo theorem (Theorem 4.2). Next, the form of an affine array access is defined (Definition 4.3). After that, a formula for the address of an array access is defined for row-major addressing (Definition 4.4). Next, the condition for bank disambiguation of an array access is expressed as a mathematical formula (Theorem 4.5). Then, for the mathematical formula for bank disambiguation to be satisfied, a formula for the step-size required after unrolling is derived (Theorem 4.6). Next, the unroll factor implied by the step-size required after unrolling, is shown to result in the minimum code-size increase among unrolls that provide bank disambiguation (Theorem 4.7). Finally, for each loop in the loop nest containing the affine access, the formula for the actual unroll factor is derived, such that the required step-size after unrolling is attained (Theorem 4.8).

In all the proofs that follow, all variables and constants introduced are integers. The proof begins by supplying two supporting theorems, theorems 4.1 and 4.2, involving modular arithmetic.

Theorem 4.1 requires Lemmas 1 and 2; stated below without proof.

Lemma 1 *Let $X \geq 0, N \geq 1$. From the definition of* gcd *(greatest common divisor), $N = \gcd(N, X)p_1$, and $X = \gcd(N, X)p_2$ where $p_1, p_2 \geq 1$ are relatively prime.*

□

Lemma 2 *Let $X \geq 0, N \geq 1$, and p_1, p_2 are as defined in Lemma 1. Then $\mathrm{lcm}(N, X) = p_1 X = p_2 N$.*

□

Theorem 4.1 (Product modulo theorem) *Assume $D, X \geq 0$ and $N \geq 1$. Then $\min\{D \mid DX \bmod N = 0\} = N/\gcd(N, X)$.*

Proof From lemma 1, $p_1 = N/\gcd(N,X)$; hence showing $\min\{D \mid DX \bmod N = 0\} = p_1$ is adequate.

From the definition of minimum[2]:

$\min\{D \mid DX \bmod N = 0\} = p_1 \iff$
$(p_1 X \bmod N = 0 \text{ and } (p_3 X \bmod N = 0 \text{ implies } p3 \geq p1))$

Hence proving $\min\{D \mid DX \bmod N = 0\} = p_1$ is equivalent to proving two claims: first,

$$p_1 X \bmod N = 0;$$

and second,

$$p_3 X \bmod N = 0 \text{ implies } p_3 \geq p_1.$$

Both these claims are proven below.

First,

$$\begin{aligned} p_1 X \bmod N &= \operatorname{lcm}(N,X) \bmod N && \text{(from lemma 2)} \\ &= 0. && \text{(from definition of lcm)} \end{aligned}$$

Second, to show $p_3 X \bmod N = 0$ implies $p_3 \geq p_1$, a proof by contradiction follows. Assume there exists some $p_3 < p_1$ such that $p_3 X \bmod N = 0$. Then,

$$\begin{aligned} p_3 X &= kN \text{ for some } k > 0 && (\text{since } p_3 X \bmod N = 0) \\ &\geq \operatorname{lcm}(N,X) && \text{(from definiion of lcm)} \\ &\geq p_1 X. && (\text{from } [\mathbf{1}]) \end{aligned}$$

Cancelling X, we get $p_3 \geq p_1$, which violates the contrary assumption $p_3 < p_1$, thus proving the result.

□

Another theorem involving integer modulo arithmetic follows.

Theorem 4.2 (Sum-of-products modulo theorem) *Let $n \geq 1$, $N \geq 1$, and $b_i \geq 0$ for $1 \leq i \leq n$. Then, for all $k_i \geq 0$ for $1 \leq i \leq n$,*

$$(k_1 b_1 + \cdots + k_n b_n) \bmod N = 0 \quad \textit{implies} \quad b_i \bmod N = 0 \textit{ for all } i, 1 \leq i \leq n.$$

Proof (by contradiction) Assume that there exists $i, 1 \leq i \leq n$ such that $b_i \bmod N \neq 0$, but for all $k_i, 1 \leq i \leq n$, $(k_1 b_1 + \cdots + k_n b_n) \bmod N = 0$.

Since $(k_1 b_1 + \cdots + k_n b_n) \bmod N = 0$ is true for all k_i, it is true for $k_i = 1$ and $k_j = 0, j \neq i$, for some particular i. This yields

[2]The notation '$\iff$' denotes "if and only if"

$b_i \bmod N = 0$.

The above is a contradiction, as we started with $b_i \bmod N \neq 0$, thus proving the result. □

The form of an affine array access assumed in the rest of the section is as follows:

Definition 4.3 *The* ***representation of an affine array access****, assuming a d-dimensional affine array access in a k-deep loop nest is*

$$A[(\textstyle\sum_{j=1}^{k} c_{1,j}v_j) + c_{1,k+1}, \quad \ldots \quad , (\textstyle\sum_{j=1}^{k} c_{d,j}v_j) + c_{d,k+1}].$$

□

In the rest of the section, let $address(val_1, \ldots, val_k)$ be defined as the address of the affine function in definition 4.3, at index variable values $v_j = val_j$ $(1 \leq j \leq k)$, assuming row-major array layout.

The following definition gives the formula for the address of an array reference assuming a row-major layout for the array.

Definition 4.4 *For a row-major array layout,* $address(val_1, \ldots, val_k) =$

$$\begin{aligned}
\&A \quad & + (\ldots((c_{1,1}MAX_2 + c_{2,1})MAX_3 + c_{3,1}) + \cdots + c_{d,1})val_1 \\
& \vdots \\
& + (\ldots((c_{1,k}MAX_2 + c_{2,k})MAX_3 + c_{3,k}) + \cdots + c_{d,k})val_k \\
& + (\ldots((c_{1,k+1}MAX_2 + c_{2,k+1})MAX_3 + c_{3,k+1}) + \cdots + c_{d,k+1})
\end{aligned}$$

where $\&A$ *is the base address of array* A*. The formula follows from the definition of row-major array addressing.*

□

The theorem below derives the condition for memory bank disambiguation for an affine function access of the form in definition 4.3. Section 1.3 states the condition for disambiguation of any memory-reference instruction: *that the memory-reference instruction access the same compile-time determinable bank in every dynamic instance.* This condition is specialized to the affine functions below. Theorem 4.5 assumes that array A is low-order interleaved among N banks[3].

For all $j, 1 \leq j \leq k$, let l_j be the lower bound of the jth loop nest from the outside enclosing an affine access.

[3]Low-order interleaving is a data layout in which array elements are interleaved among the different banks in a round-robin manner, beginning at bank 0. That is, element A[i] is allocated to bank $i \bmod N$.

Theorem 4.5 (Disambiguation condition) *Suppose all the loop nests enclosing an affine access are unrolled, such that the jth loop nest $(1 \le j \le k)$ has a step size of D_j after unrolling. Then, the condition for memory disambiguation of that access is*

$$address(l_1, \ldots, l_k) \bmod N = address(l_1 + m_1 D_1, \ldots, l_k + m_k D_k) \bmod N$$

for all positive integral values of m_j $(1 \le j \le k, forall\ m_j \ge 0)$.

Proof The address of the affine function in the first iteration of all the loops is $address(l_1, \ldots, l_k)$. Since the array A is assumed to be low-order interleaved across banks with $A[0]$ on bank 0, it follows that the bank number on which $address(l_1, \ldots, l_k)$ resides is the low-order bits of the address, *i.e.*,

$$address(l_1, \ldots, l_k) \bmod N. \qquad \textbf{[3]}$$

Similarly, the bank number for any later iteration is represented as follows. Any later iteration increments the jth loop index variable value by $m_k D_k$ where m_k is the number of iterations of the unrolled loop, and D_k is its step size. Hence, the address of any later iteration is represented as

$$address(l_1 + m_1 D_1, \ldots, l_k + m_k D_k) \bmod N. \qquad \textbf{[4]}$$

For disambiguation, it is necessary and sufficient that the bank number for all iterations be the same, and hence the same as in the first iteration. This condition for disambiguation is the same as equating [**3**] and [**4**], yielding

$$address(l_1, \ldots, l_k) \bmod N = address(l_1 + m_1 D_1, \ldots, l_k + m_k D_k) \bmod N.$$

□

The following theorem derives the minimum value of D_j, the step size after unroll, needed for disambiguation of the affine access in consideration.

Theorem 4.6 (Formula for D_j) *To ensure that the condition for disambiguation given in theorem 4.5 is satisfied, the minimum value of D_j $(1 \le j \le k)$ must be*

$$D_j = N \ / \ \gcd\ \left(N,\ \sum_{i=1}^{d} \left(c_{i,j} \prod_{l=i+1}^{d} MAX_l \right) \right)$$

Any multiple of the above value for D_j also satisfies the condition.

Proof The condition from theorem 4.5 is

$$address(l_1, \ldots, l_k) \bmod N = address(l_1 + m_1 D_1, \ldots, l_k + m_k D_k) \bmod N$$

for all positive integral values of m_j $(1 \le j \le k, \text{forall } m_j \ge 0)$. Using definition 4.4, both sides of this condition are expanded. First, the left-hand side of the condition is

$$(\&A\ +\ (\ldots((c_{1,1} MAX_2 + c_{2,1}) MAX_3 + c_{3,1}) + \cdots + c_{d,1})\ l_1$$

$$
\begin{aligned}
&\vdots \\
&+ (\ldots((c_{1,k} MAX_2 + c_{2,k}) MAX_3 + c_{3,k}) + \cdots + c_{d,k})\ l_k \\
&+ (\ldots((c_{1,k+1} MAX_2 + c_{2,k+1}) MAX_3 + c_{3,k+1}) + \cdots + c_{d,k+1}))\ \mathrm{mod}\ N.
\end{aligned}
$$

Second, the right-hand side of the condition is

$$
\begin{aligned}
(\&A\ &+ (\ldots((c_{1,1} MAX_2 + c_{2,1}) MAX_3 + c_{3,1}) + \cdots + c_{d,1})\ (l_1 + m_1 D_1) \\
&\vdots \\
&+ (\ldots((c_{1,k} MAX_2 + c_{2,k}) MAX_3 + c_{3,k}) + \cdots + c_{d,k})\ (l_k + m_k D_k) \\
&+ (\ldots((c_{1,k+1} MAX_2 + c_{2,k+1}) MAX_3 + c_{3,k+1}) + \cdots + c_{d,k+1}))\ \mathrm{mod}\ N.
\end{aligned}
$$

The left-hand side of the equality is of the form a mod N, and right-hand side is of the form b mod N. If a mod $N = b$ mod N, it follows that $(b - a)$ mod $N = 0$. Substituting in a and b yields

$$
\begin{aligned}
&((\ldots((c_{1,1} MAX_2 + c_{2,1}) MAX_3 + c_{3,1}) + \cdots + c_{d,1})\ m_1 D_1\ + \\
&\vdots \\
&(\ldots((c_{1,k} MAX_2 + c_{2,k}) MAX_3 + c_{3,k}) + \cdots + c_{d,k})\ m_k D_k)\ \mathrm{mod}\ N = 0.
\end{aligned}
$$

The above desired condition is exactly the form of the required condition in theorem 4.2, with m_i here equal to k_i in the theorem, and k here equal to n in the theorem. Hence the implied expression in that theorem is true for disambiguation, which translates in this instance to

$$D_j(\ldots((c_{1,j} MAX_2 + c_{2,j}) MAX_3 + c_{3,j}) + \cdots + c_{d,j})\ \mathrm{mod}\ N = 0 \quad (1 \le j \le k),$$

i.e.,

$$D_j \left(\sum_{i=1}^{d} \left(c_{i,j} \prod_{l=i+1}^{d} MAX_l \right) \right) \mathrm{mod}\ N = 0\ (1 \le j \le k).$$

Using theorem 4.1, the minimum value of D_j satisfying this condition is:

$$D_j = N \ / \ \gcd \left(N,\ \sum_{i=1}^{d} \left(c_{i,j} \prod_{l=i+1}^{d} MAX_l \right) \right)$$

Any multiple of the above value for D_j also satisfies the condition.

□

The following theorem shows that the minimum value of D_j derived in theorem 4.6 minimizes the overall code size for unrolling schemes that provide disambiguation.

Theorem 4.7 (Minimum code size) *The value of D_j in theorem 4.6 minimizes the overall code size of the entire loop nest unrolled appropriately for providing disambiguation.*

Proof The overall code size for the unrolled loop nest is proportional to $D_1 \dots D_k$, *i.e.*, the product of the unrolled step sizes. From theorem 4.6, the given value of D_j $(1 \leq j \leq)$ is minimum for disambiguation for each j, independent of the values of D_j at other j. Hence the product of D_j's is minimized when D_j for each j is individually minimized, as was done in theorem 4.6.

□

The following theorem derives the final result, *i.e.*, the value of the unroll factor U_j, in terms of step size after unroll D_j.

Theorem 4.8 (Unroll factor formula) *In order to attain the value of D_j in theorem 4.6, we need to unroll the* jth *loop nest* $(1 \leq j \leq k)$ *by a factor U_j given by*

$$U_j = \mathrm{lcm}(D_j, s_j) \ / \ s_j.$$

Proof By definition:

D_j is the desired step size of the jth loop in the loop nest unrolled for disambiguation.

s_j is the step size of the jth loop in the loop nest before unrolling.

If D_j is a multiple of s_j, then the unroll factor is D_j/s_j. In general, it may not be. Attainable unrolled step sizes are

$$s_j \text{ or any multiple.} \qquad [\mathbf{5}]$$

For disambiguation, from theorem 4.6, unrolled step sizes must be

$$D_j \text{ or any multiple.} \qquad [\mathbf{6}]$$

Combining [**5**] and [**6**], the lowest attainable step size that results in disambiguation is $\mathrm{lcm}(D_j, s_j)$. The unroll factor is the unrolled step size divided by the initial step size, *i.e.*,

$$U_j = \mathrm{lcm}(D_j, s_j) \ / \ s_j.$$

□

4.4 Code growth: bounds and the padding optimization

This section examines the increase in code size implied by the modulo unrolling. Code growth is an undesirable side-effect of modulo unrolling; first, upper bounds on how much the code size can grow in the worst case are derived. Second, the padding optimization is described; it reduces the code growth for most cases seen in practice. Finally, an example demonstrates the application of the modulo unrolling formulas, both with and without the padding optimization.

Bounds on unroll factors

Unrolling incurs the cost of increased code size. To establish a bound, we show that the unroll factor U_j derived in Theorem 4.8 is provably at most N, the number of banks. Consider that D_j is at most N as D_j is computed by an expression that is N divided by a positive, non-zero integer. Further, since U_j is given by $\mathrm{lcm}(D_j, s_j)/s_j$, U_j is no more than D_j, because the lcm of two numbers is always at most their product. Hence U_j is at most N.

In the worst case, since all the k loop in the loop nest may be unrolled N ways, the overall code growth is at most a factor of N^k. For k $\geq$2, N^k can be large. In practice, however, for most affine functions, the overall code growth is often limited to N irrespective of k by using the padding optimization, discussed later in this section.

A final observation regarding code growth is that the decision of whether or not to modulo unroll any one nested loop does not affect the rest of the code. In particular, if the code growth from modulo unrolling is deemed excessive for any one nested loop, that nested loop is not unrolled; the accesses in it are made non-disambiguated, and the rest of the code is unaffected.

Padding Optimization

For many affine functions that occur in practice, a simple optimization enables us to restrict the overall code growth, as well as greatly simplify the code generation. This optimization is the *padding optimization*, which involves padding the last array dimension size to be a multiple of N for all arrays. To see how, first we derive a simpler expression for D_j than the one in theorem 4.8, in the case when the padding optimization is performed.

Corollary 4.9 *In theorem 4.8, if the last dimension of the array (MAX_d) is padded to the next higher multiple of N, then the expression for D_j simplifies to*

$$D_j = N \ / \ \gcd\ (N,\ c_{d,j}).$$

Proof From the expansion of the expression for D_j in theorem 4.6, we get

$$D_j = N \ / \ \gcd\ (N, ((\ldots((c_{1,j}MAX_2 + c_{2,j})MAX_3 + c_{3,j}) + \cdots)MAX_d + c_{d,j})).$$

The above equation is of the form

$$D_j = N \ / \ \gcd\ (N,\ (XMAX_d + c_{d,j})).$$

where X is some integer. If MAX_d is a multiple of N, it follows that irrespective of the value of X, the gcd expression simplifies to $\gcd(N, c_{d,j})$, which yields the result. It can be shown that since the value of X does not matter for this result, the result holds for cases when all but the last dimension of the array reference are not affine.

□

We define the following class of array references that are shown to benefit from the padding optimization.

Definition 4.10 (Simple-index last dimension) *The simple-index last dimension class of array references are those whose index expression in the last array dimension is of the form $c_1 * i + c_2$, where i is any loop induction variable and c_1, c_2 are any integer constants. The array index expressions other than for the last dimension are unrestricted, and need not even be affine.*

Most affine functions that occur in real programs are of the simple-index last dimension class. Some references that have non-affine expressions in all but the last array dimension are also in this class. The following theorem shows that for this class, at most one of the enclosing loops is unrolled by modulo unrolling.

Theorem 4.11 *Consider an array reference of the simple-index last dimension class. If the array accessed by the reference has its last dimension padded to a multiple of N, then at most one of its enclosing loops is unrolled. That is, the U_j values for the other loops will automatically be 1.*

Proof The expression for D_j in this case is obtained from corollary 4.9 as

$$D_j = N \;/\; \gcd\;(N,\;\; c_{d,j}).$$

Recall that $c_{d,j}$ is the index of the jth loop induction variable in the last array dimension. Since the reference is a simple-index in the last dimension, all but one of the $c_{d,j}$'s for different j are zero. For all these j values with $c_{d,j} = 0$, it follows that $D_j = N/\gcd(N, 0) \;=\; 1$. Hence, U_j, the unroll factor, is also 1.

□

Theorem 4.11 further tightens the code-size growth bound of N^k derived at the beginning of section 4.4, where N is the number of memory banks, and k is the number of loops in the enclosing loop nest of the affine reference. Using the padding optimization, however, theorem 4.11 shows that the code-size growth is no more than N for array references that are in the simple-index last dimension class, irrespective of k, thus tightening the bound.

In some cases the padding optimization fails to bound the overall code growth to N. Such cases include those where the affine functions are not simple index functions, as well as cases where the loop nest contains multiple simple index functions that induce unrolls on different loops of the loop nest. For cases where the code growth is more than a factor of N, if the code growth is deemed excessive, the array accesses are executed on the dynamic network, thus avoiding any code growth. The dynamically executed accesses, however, do not interfere with the disambiguation of other accesses.

Example of applying modulo unrolling

As an example of how modulo unrolling is used to automatically compute the unroll factors, consider figure 4.2. Figure 4.2(a) shows a code fragment from Tomcatv, one of the Spec92 benchmarks. Figure 4.2(b) shows the unroll factors computed using modulo unrolling for the $X[I][J]$ and $X[I][1]$ accesses on different rows, for both I and J loops. The last row shows the overall unroll factors. The second column shows the expressions for unroll factors using the formula in Theorem 4.8. Assuming $N = 8$ and array sizes for the X and Y arrays being 29×29, the third and fourth columns show the unroll factors with and without the padding optimization. In the fourth column, the last dimension size JMAX = 29 is assumed to be padded to 32, the next multiple of N. Using the unroll factor formula, the unroll factors for the $X[I][J]$ access for the I and J loops are shown as 1 and 8 after padding. The unroll factors would have been larger (8 and 8) without padding. If the unroll factors induced by all the accesses are similarly computed (not shown), and the lcm for each loop in the loop nest taken, the overall unroll factors in the last row results.

4.5 An additional transformation

This section describes an additional transformation needed after modulo unrolling for bank disambiguation of loops having an unknown lower bound and a non-unit step size.

After the code is unrolled by the factors dictated by Theorem 4.8, each affine array access refers to the same bank in every dynamic instance. In addition, in most cases after unrolling, the bank numbers of the accesses are compile-time constants. For a loop with an *unknown lower bound and a non-unit step size*, however, the repeating pattern of bank numbers may depend on the lower bound. As an example, consider the code in figure 4.3. When the lower bound l is 0, the banks referenced by successive accesses is 0, 2, 0, 2, ..., but if l is 2 the pattern changes to 1, 3, 1, 3, ... As a result, bank disambiguation is not possible.

To allow bank disambiguation for a loop with unknown loop bounds and non-unit step size, a switch statement is needed in the output code. The switch has D_j/U_j cases and is made on the value of $l \bmod (D_j/U_j)$, each case executing the original loop unrolled by a factor U_j but with different patterns of bank numbers. The resulting code for the example in figure 4.3 is shown in figure 4.4. For this example, $D_j = 4$ and $U_j = 2$.

```
real X[IMAX][JMAX], Y[IMAX][JMAX]

for I=2 to M-1 do
    for J=2 to M-1 do
        X[I][J] = 0.9 * X[I][1]
        Y[I][J] = 0.9 * ((1.0 - X[I][1]) * Y[1][J] + X[I][1] * Y[M][J])
endfor
```

(a)

ACCESS		*UNROLL FACTORS*		
		Abstract Expression	Without padding (JMAX = 29)	With padding (JMAX padded to 32)
$X[I][J]$	Loop I	$D_I = U_I = N/\gcd(N, 1 * \text{JMAX} + 0 * 1)$	$8/\gcd(8, 29) = 8$	$8/\gcd(8, 32) = 1$
	Loop J	$D_J = U_J = N/\gcd(N, 0 * \text{JMAX} + 1 * 1)$	$8/\gcd(8, 1) = 8$	$8/\gcd(8, 1) = 8$
$X[I][1]$	Loop I	$D_I = U_I = N/\gcd(N, 1 * \text{JMAX} + 0 * 1)$	$8/\gcd(8, 29) = 8$	$8/\gcd(8, 32) = 1$
	Loop J	$D_J = U_J = N/\gcd(N, 0 * \text{JMAX} + 0 * 1)$	$8/\gcd(8, 0) = 1$	$8/\gcd(8, 0) = 1$
Overall	Loop I		8	1
	Loop J		8	8

(b)

Figure 4.2: Modulo unrolling for code fragment from Tomcatv. (a) Code fragment. (b) Unroll factors computed for two of the accesses for $N = 8$ and X and Y array sizes being 29×29. The unroll factors are shown with and without the padding optimization. The last row shows the overall unroll factor computed to be the lcm of all 7 accesses.

for i=l to 99 step 2 do
  A[i] = $\cdots$
endfor

Figure 4.3: Sample loop with unknown lower bound and non-unit step size.

```
switch (l mod 2)
  case 0:   for i=l to 99 step 4 do
               A[i] = ···          //bank 0
               A[i+2] = ···        //bank 2
            endfor
  case 1:   for i=l to 99 step 4 do
               A[i] = ···          //bank 1
               A[i+2] = ···        //bank 3
            endfor
endswitch
```

Figure 4.4: Code transformed for disambiguation (4 bank system) for example in figure 4.3. The original loop has an unknown lower bound and non-unit step size, resulting in a bank access pattern that depends upon the lower bound value. The transformed code has a switch statement with D_j / U_j cases, the switch made on $l$mod (D_j/U_j). Each case has a compile-time-known bank access pattern.

4.6 Affine code generation

Once loops are unrolled and the additional transformation in section 4.5 is performed when required, each affine access refers to the same bank in all dynamic instances. This section outlines how the constant bank numbers, and the expressions for local offsets within the banks, are actually computed.

Code generation effectively distributes a single array of S elements in the original program across the banks, so that each bank has an array of size $\lceil S/N \rceil$. Using low-order interleaving, the bank number of an access is its global offset modulo N, and the local offset is the global offset divided by N. When the last dimension is padded, as is done because of the padding optimization in section 4.4, the bank number is simply the last dimension modulo N. In addition, the local offset is obtained by replacing the last dimension index with the index divided by N.

Strip mining

While the observations in section 4.6 can be used to generate code directly, code generation is automated by strip-mining the last dimension by N and strength-reducing the divide operations. This process of strength-reduction following strip mining is done following the method for strength-reduction in [38]. Strip mining replaces the last dimension by itself divided by N, and it adds a new dimension at the end with the index being the original last dimension index mod N. The division expressions are strength-reduced in all cases, and the mod expressions representing bank numbers are reduced to constants using compiler knowledge of the modulo values of loop variables combined with modulo arithmetic [39].

Startup and cleanup code

Unrolling always generates cleanup code after the unrolled loop if the number of iterations is not a multiple of the unroll factor. In addition, unrolling in the modulo unrolling method generates startup code when the lower bound is unknown. The startup code ensures that the main loop is started with the induction variable value being the next higher multiple of N, thus making the bank numbers known inside the main loop.

```
idiv4 = 0;
for i=0 to 99 step 4 do
   A[idiv4][0] = ···
   A[idiv4][1] = ···
   A[idiv4][2] = ···
   A[idiv4][3] = ···
   idiv4++
endfor
```

Figure 4.5: Bank-disambiguated code for example in figure 1.7.

Figure 4.5 shows the final result of bank disambiguation on the original code in figure 1.7 for a four-bank Raw machine. The code is first unrolled by $N = 4$ and the last array dimension is strip mined by $N = 4$. The division expression is strength reduced to the variable *idiv*4. The new last dimension in figure 4.5 represents the bank numbers, which have been reduced to the constants 0, 1, 2 and 3. The bank access pattern in the transformed loop is 0, 1, 2, 3, 0, 1, 2, 3 ... as in the original code, except that now each access always refers to the same bank. In the Raw compiler, the transformed code is finally mapped by the space-time scheduler to the Raw executable.

4.7 Other optimizations for array accesses

This section outlines two optimizations for array accesses on Raw in addition to modulo unrolling: dependence elimination and the array-permutation transformation.

Dependence elimination

Dependence edges are introduced between memory instructions that the compiler can either prove to refer to the same location, or cannot prove to refer to different locations.

Unnecessary dependence edges restrict ILP, since they imply sequentiality of accesses and thus restrict scheduling freedom. For scientific codes containing affine array accesses, three simple rules suffice to disambiguate most memory instructions that can be disambiguated. First, pointer analysis proves that instructions referring to different unaliased arrays are memory-independent. Second, memory instructions that are found to refer to different banks by modulo unrolling are always memory-independent. Finally, even among memory instructions referring to the same bank, instructions belonging to the same uniformly generated set[4], and differing by a non-zero constant are memory--independent.

Array-permutation transformation

Sometimes modulo unrolling unrolls the outer loop in a loop nest and not the inner loop. Unrolling only the outer loop is ineffective in terms of exposing ILP within basic blocks, because the basic blocks are very small. The array-permutation transformation is a transformation that replaces array references inducing outer loop unrolling by references to a permuted array such that the new references induce unrolling on the inner loop. A permuted array is one that has the same number and size of dimensions as the original array, but in a permuted order. When all the loops in a program request the same permutation for an array, the original array is replaced by the permuted array. When different loops request conflicting permutations, it may be profitable to copy one permuted array to another in between loops.

The array-permutation transformation is currently performed by hand in places where it might be profitable. The transformation can be automated by discovering permutations desired, and by using a cost model to determine when copying is profitable.

4.8 Summary

This chapter describes modulo unrolling, the second of our two methods for bank disambiguation. Modulo unrolling improves upon the first method, equivalence-class unification, by exploiting memory parallelism within arrays, while providing bank disambiguation for affine array accesses. The method for modulo unrolling is first stated, then the formula for the unroll factor stated is derived. The padding optimization helps control the code size increase, and simplifies code generation. Modulo unrolling is shown to work for arbitrary affine functions, imperfectly nested loops and unknown loop bounds. Most importantly, modulo unrolling continues to work for loops with affine accesses even when they also contain non-affine accesses. This behavior is unlike many affine-based parallelization schemes such as those for multiprocessor and vector compilers.

[4]Two affine memory instructions are in the same uniformly generated set if they access the same array, and their index expressions differ by at most a constant. For example, $A[i]$ and $A[i+2]$ are in the same uniformly generated set, but $A[i]$ and $A[i+j]$ are not [40].

Chapter 5

Non-disambiguated or dynamic accesses

This chapter describes the mechanisms Maps provides for correctly and efficiently handling non-disambiguated memory-reference instructions on bank-exposed architectures. Non-disambiguated accesses are those that are not disambiguated to a particular bank by bank disambiguation. Bank disambiguation disambiguates every memory instruction that is known to access the same compile-time-known bank in every dynamic instance. Non-disambiguated references, by definition, may go to different banks in different dynamic instances; hence non-disambiguated references cannot be scheduled at compile-time. Since compiler-scheduling is not possible, a separate run-time routed network, called a *dynamic network*, must be provided on the bank-exposed architecture to complete non-disambiguated references. The dynamic network may be either a hardware-managed centralized network, or a distributed network under software control.

This chapter is organized as follows. Section 5.1 begins by showing the need for dynamic accesses. Section 5.2 illustrates how Maps enforces different kinds of memory dependences. Section 5.3 shows how the kind of dynamic network used influences how dependences are enforced. It also shows why it is a challenge to efficiently enforce dependences on distributed dynamic networks that do not themselves provide dependence or timing guarantees. The rest of this chapter is devoted to optimizations possible on a distributed dynamic network, in case one is used. Section 5.4 describes *software serial ordering* (SSO), an optimization method for aggressively overlapping dynamic access latencies on a distributed network, while correctly enforcing dependences between dynamic accesses. Section 5.5 describes how dependences across scheduling units are maintained while using SSO, irrespective of what kind of dynamic network is used. Section 5.6 describes optimizations that improve performance by reducing the amount of dependences that need to be enforced. Two optimizations, independent epochs and memory updates, are presented. The Raw machine uses a distributed dynamic network, and thus can benefit from SSO and its optimizations. Section 5.7 describes future work. Section 5.8 summarizes the chapter.

From this point, the terms 'non-disambiguated' and 'dynamic' are used interchange-

ably, and so are the terms 'disambiguated' and 'static'. Both sets of terminologies are maintained as a distinction between concept and implementation: non-disambiguated accesses are implemented as dynamic accesses; disambiguated accesses are implemented as static accesses.

5.1 Uses for dynamic references

This section motivates the need for non-disambiguated accesses by showing that although they are slower than disambiguated accesses, selective use of non-disambiguated accesses can improve performance in certain cases.

A compiler can bank disambiguate all accesses through equivalence-class unification alone, and modulo unrolling helps improve memory parallelism further. There are several reasons, however, why it may be undesirable to disambiguate all accesses. Some reasons are listed here:

- Modulo unrolling sometimes requires unrolling of more than one dimension of multi-dimensional loops. This unrolling can lead to excessive code expansion. To reduce the unrolling requirement, some accesses in these loops can be made dynamic.

- Bank disambiguation may sometimes be achieved at the expense of memory parallelism. For example, indirect array accesses of the form $A[B[i]]$ cannot be disambiguated unless the array $A[]$ is placed entirely on a single bank. This placement, however, yields no memory parallelism for $A[]$. Instead, Maps can choose to forgo bank disambiguation and distribute the array. Indirect accesses to these arrays would be implemented dynamically, which yields better parallelism at the cost of higher access latency.

- Dynamic accesses can improve performance by not destroying compile-time memory parallelism in critical parts of the program. Without dynamic accesses, arrays with mostly affine accesses but a few irregular accesses, even in non-critical portions, would have to be mapped to one bank, thus losing all potential memory parallelism to the arrays.

- As described in section 3.3, dynamic accesses can increase the resolution of equivalence class unification. A few isolated "bad references" may cause pointer analysis to yield very few equivalence classes. By selectively removing these references from disambiguation consideration, more equivalence classes can be discovered, enabling better data distribution and improving memory parallelism. The misbehaving references can then be implemented as dynamic accesses.

For these reasons, it is important to have a good fall-back mechanism for dynamic references. More important, such a mechanism must integrate well with the static (disambiguated) mechanism. The next section explains how these goals are accommodated.

For a given memory access, the choice of whether to use a static or a dynamic access is not always obvious. Because of the significantly lower overhead of static accesses, the current Maps system makes most accesses static by default, with one exception. Arrays with any affine accesses are always distributed, and two types of accesses to those arrays are implemented as dynamic accesses: non-affine accesses, and affine accesses that require excessive unroll factors for bank disambiguation. Automatic detection of other situations which can benefit from dynamic accesses has not been implemented. However, section 9 shows two programs, Unstructured and Moldyn, whose performance can be improved when dynamic accesses are selectively used.

5.2 Enforcing dependences

The issue of implementing dynamic accesses efficiently is closely tied to the enforcing of possible dependences between memory accesses. This section describes how dependences between different combinations of static and dynamic references are enforced. We show that dynamic-dynamic dependences are the hardest to enforce efficiently.

Dependence pairs on a software-exposed architecture can be classified into three types: those between two static accesses, those between a static and a dynamic access, and those between two dynamic accesses. The first two types can be efficiently enforced in a straightforward manner as follows. First, dependences between static accesses are easily enforced. References mapped to different memory banks are necessarily non-conflicting. For references mapped to the same bank, the compiler need only ensure that their order in the final code is the same as in the original source. Since static accesses are fast, there is little opportunity or need for overlap of latency. Second, a dependence between a static and a dynamic accesses is also enforced in a straightforward manner. A static synchronization message can be used to communicate the completion of the first access to the initiation of the dependent access. Since static messages are fast, this synchronization has little overhead. If a dynamic store is followed by a dependent static load, this synchronization requires an extra dynamic store acknowledgment message at the completion of the store, so that a clear completion point for the dynamic store exists in the code.

For dependences between pairs of dynamic memory references, however, a straightforward implementation that serializes the references is slow. Dynamic access latencies are long compared to static access latencies. Complete serialization fails to overlap the dynamic access latencies; hence techniques to handle dynamic latencies focus on trying to overlap the dynamic latencies as much as possible while maintaining correctness.

5.3 Enforcing dependences between dynamic accesses

This section shows how enforcing dependences between dynamic accesses depends upon the kind of dynamic network used. Enforcing dependences between pairs of dynamic accesses can be very expensive; fortunately, optimizations are possible. The costs and optimizations possible depend upon the nature of the dynamic network itself. Finally, the section shows the challenge in enforcing dependence on a distributed network that does not itself provide any dependence or timing guarantees.

Different kinds of dynamic networks

There are many different kinds of networks [9] that can be used on a bank-exposed machine. Dynamic networks connect the tiles on bank-exposed architectures; possible topologies include buses, linear arrays, meshes, trees, hypercubes, butterflies, and networks centralized at a serial arbitrator.

For the purposes of enforcing dependences, we *classify dynamic networks into three types* as follows. The *first* class of networks are those that enforce dependences in hardware. In this class of network, the hardware ensures that accesses that are issued in program order commit to memory in order. In such networks the task of overlapping latency falls to the hardware, and there is no role for the compiler in this aspect. Most microprocessor memory buses with associated arbitration logic fall in this class. The *second* class of networks are those in which the hardware does not guarantee that accesses that issue in program order commit in order, yet, the network guarantees the timing between the issue and commit of a request message, irrespective of network contention. Such networks include full crossbars, whether implemented as single-stage or multi-stage networks. Such networks, *i.e.*those having guaranteed timing between issue and commit of memory references, allow the compiler to enforce dependence by ensuring two properties: one, by statically scheduling requests such that only one request reaches each bank per cycle, and two, by ensuring that the requests reach the banks in program order, even if issued out of order. Such static scheduling of the network is possible using methods similar to those in the Raw space-time scheduler [25] for the static network.

The third kind of dynamic network is one in which not only does the hardware not guarantee dependence, but in addition the timing between the issue and commit of a request message is not guaranteed, and depends upon the network contention. Most distributed-memory multiprocessor networks, such as meshes, trees, and buses are in this class of network. The Raw machine explores such a network with no timing guarantees. The motivation for Raw to explore such a network is that networks with no dependence checking and no timing guarantees is two-fold. First, such networks are more scalable, as they have no centralized dependence-checking hardware. Second, such networks require less wires than guaranteed-timing networks, and thus are easier to implement.

Enforcing dependences in the compiler

In networks with no dependence checking in hardware and no timing guarantees, the task of ensuring dependence falls to the compiler. The lack of dependence guarantees opens up new challenges in the compiler; however, the scalability of distributed network implementations opens up new opportunities as well. If compiler techniques are effective, they can exploit far more parallelism than can centralized arbitration logic. The rest of this chapter is dedicated to such techniques for distributed dynamic networks.

To see the difficulty involved in enforcing dependences efficiently on networks with no hardware dependence enforcement and no timing-guarantees, we begin with an example. Consider the case of a dependence that orders a dynamic store before a potentially conflicting dynamic load. Because of the dependence, it would not be correct to issue the two requests in parallel from different tiles. Furthermore, it would not suffice to synchronize the issues of the requests on different tiles. Raw-like distributed dynamic networks have no timing guarantees: even if the memory operations are issued in correct order, they may still be delivered to a memory bank in incorrect order. One obvious solution is complete serialization as shown in figure 5.1(a), where the later memory reference cannot initiate until the earlier reference is known to complete. Complete serialization, however, is expensive because it serializes the slow round-trip latencies of the dynamic requests.

Section 5.4 presents a new technique called software serial ordering to optimize beyond complete serialization. Software serial ordering depends upon the properties of the distributed network used. In particular software serial ordering leverages the *in-order* messaging property provided by many dynamic networks. The in-order property states that if two or more messages are sent between the same source and destination, they appear at the destination tile in the same order as they were launched at the source. Software serial ordering is described in section 5.4.

5.4 Software serial ordering

This section describes a new technique called *software serial ordering* (SSO) to efficiently ensure dependences between dynamic accesses on networks that offer no hardware dependence enforcement and no timing guarantees. It then illustrates SSO using an example.

Figure 5.1(b) illustrates SSO. SSO leverages the in-order delivery of messages on the point-to-point network between any source-destination pair of tiles. SSO works as follows. Each equivalence class is assigned a *turnstile* node. The role of the turnstile is to serialize the request portions of the memory references in the corresponding equivalence class. The address computations for the different accesses in one class are not serialized, only the requests are serialized. Moreover, the address computations communicate their addresses to turnstiles using register-level (static) messages. Once memory references go through the turnstile in the right order, correct behavior is ensured from three facts. First, requests destined for different tiles must necessarily refer to different memory locations, so no memory dependence needs to be enforced. Second, requests destined for the same tile are delivered in order by the dynamic network, as guaranteed by the network's in-order delivery property. Finally, the memory tile handles requests in the order they are delivered.

In order to guarantee correct ordering of processing of memory requests, serialization is inevitable. SSO serializes only the memory requests, and allows the exploitation of

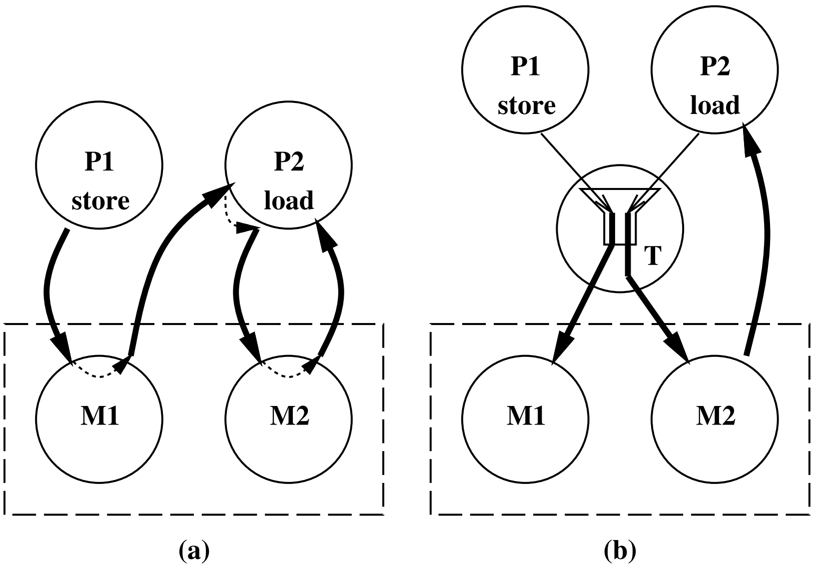


Figure 5.1: Two methods for enforcing dependences between dynamic accesses. P1 and P2 are processing nodes initiating two potentially conflicting dynamic requests; both diagrams illustrate an instance when the two requests don't conflict. M1 and M2 are the destinations of the memory requests. The light arrows are static messages, the dark arrows are dynamic messages, and the dashed arrows indicate serialization. The dependence to be enforced is that the store on P1 must precede the load on P2. In (a), dependence is enforced through complete serialization. In (b), dependence is enforced through software serial ordering. T is the turnstile node. The only serialization point is the launches of the dynamic memory requests at T. The Raw tiles are not specialized; any tile can serve in any or all of the following roles, as processing node, memory node, or turnstile node.

parallelism available in address computations, latency of memory requests and replies, and processing time of memory requests to different tiles. For efficiency, software serial ordering uses the static network to handle synchronization and data transfer whenever possible. Furthermore, different equivalence classes use different turnstiles and issue requests in parallel. Interestingly, though SSO enforces dependences correctly while allowing potentially dependent dynamic accesses to be processed in parallel, it does not use a single explicit check of run-time addresses.

Example of software serial ordering

Figure 5.2 explains SSO through an example. The initial code in figure 5.2(a) has an $A[B[i]]$ reference inside a **for** loop. The reference to $A[]$ is non-affine. Assume that in order to exploit memory parallelism between affine accesses to $A[]$ elsewhere in the program, the array $A[]$ is distributed using low-order interleaving.

Figure 5.2(b) shows the code after bank disambiguation and SSO. The $B[]$ access is disambiguated using modulo unrolling; hence, the **for** loop is unrolled by a factor of 4. The $A[]$ references in the unrolled loop remain non-disambiguated, as the references are non-affine and array $A[]$ is distributed. Consequently, the 4 $A[]$ references in figure 5.2(b) are handled using SSO. The requests are made to the turnstile using compiler-scheduled communication. Next, the turnstile issues the 4 non-blocking requests to memory serially, on the distributed dynamic network. The store requests reach memory banks that are unknown at compile-time, where they are serviced, and store acknowledgments are sent back to a PE waiting for them, decided at compile-time. Finally, when all the acknowledgments are received, the control moves to the next section of code after the unrolled loop body. Correctness is ensured because request messages from the turnstile to any one memory bank arrive in program order; in-order delivery follows from the pair-wise in-order property of the network. To see how SSO works, suppose that, in figure 5.2(b), $A[B_0[i']]$ and $A[B_1[i']]$ accesses resolve at run-time to the same address (*i.e.*, they are truly dependent). Then the requests for the two accesses, from the turnstile to the bank containing the accessed location, arrive in program order, ensuring that the writes commit in order. Figure 5.2(c) shows one possible result after code generation: the turnstile in this case is placed on PE 0.

5.5 Dependences across scheduling units

This section shows why software-serial ordering on its own is not enough; a scheme for enforcing static-dynamic dependences across scheduling units is needed. *Memory barriers* are presented as one way to enforce such dependences across scheduling units. Finally, cases when memory barriers can be optimized away are described.

Ensuring dependences between static and dynamic accesses across scheduling units poses an extra challenge, irrespective of whether a hardware-managed or software-controlled dynamic network is used. A *scheduling unit* is the granularity of code on which the space-

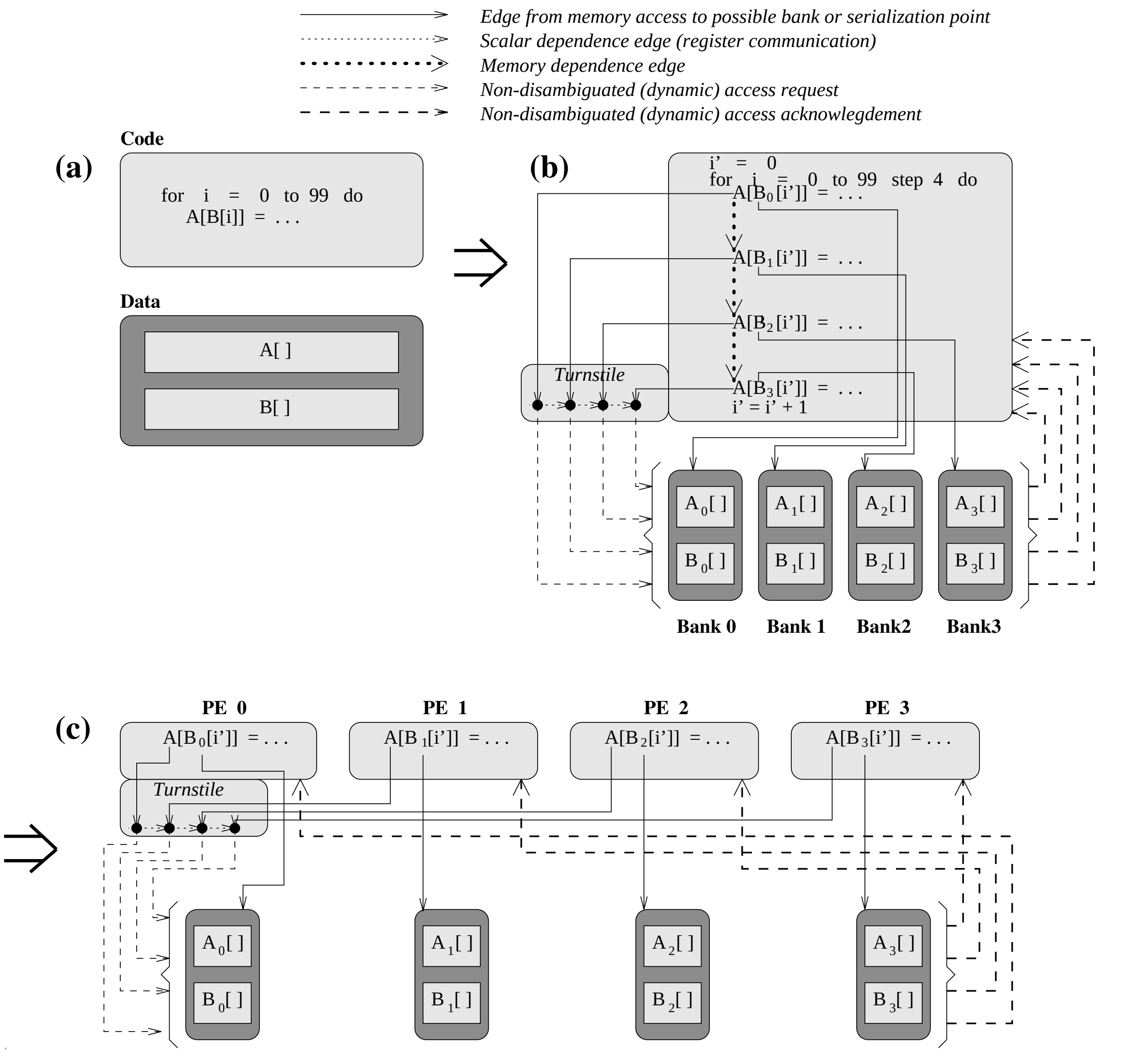


Figure 5.2: Example showing software serial ordering (SSO). (a) Initial code; (b) After bank disambiguation and SSO. $A[]$ is distributed, so $A[]$ accesses are non-disambiguated and use SSO. $B[]$ accesses are bank-disambiguated using modulo-unrolling. SSO involves serialization at the *turnstile*, which can be placed on any of the PEs. The scheduling unit (in this case the loop body) waits for the write acknowledgments before moving to the next scheduling unit. (c) After code generation.

time scheduler [25] performs instruction scheduling and code generation; the space-time scheduler is the phase following Maps in the Raw compiler. Scheduling units are basic blocks or larger; control-localization [25] extends them to forward-control-flow regions. Static-static and dynamic-dynamic dependences are enforced in a straight-forward manner: the static accesses are ordered in program order on their owner tile, and the dynamic accesses are ordered in program order on the turnstile. In contrast, static-dynamic dependences require point-to-point synchronization messages between successive accesses to ensure serialization within the same equivalence class. Point-to-point synchronization messages are easily scheduled within a scheduling unit. Across scheduling units, however, point-to-point synchronization messages cannot be compiler-scheduled because the predecessor for a scheduling unit is not known at compile-time.

Memory barriers

The Raw compiler adopts *memory barriers* as a simple solution to the problem of enforcing static-dynamic dependences across scheduling units. Run between scheduling units, memory barriers isolate different scheduling units from each other, obviating the need for pairwise synchronization between accesses in different scheduling units. A memory barrier is a software construct that waits for the completion of every reference on every tile before allowing the next scheduling unit to execute on any tile. A barrier is associated with every edge of the program's control-flow graph[1] that exits a scheduling unit, including those leading back to the start of the same unit.

A memory barrier is implemented as follows. In the predecessor scheduling unit, every tile containing static accesses, or dynamic-access returns, sends a message to a central tile upon completion of the predecessor. Next, the central tile waits until all incoming messages are received, and then broadcasts a completion message to every tile. Each tile starts executing the successor upon receiving its completion message, thus completing the barrier. Two observations regarding performance follow. First, since the barrier involves a compile-time-known communication pattern, all messages involved in the barrier are static. Second, the many-to-one and one-to-many communication patterns in a barrier are implemented as trees in case a distributed network is used, reducing the barrier overhead for when the number of tiles is large.

Figure 5.3 depicts the transition between two scheduling units without and with a barrier. Figure 5.3(a) shows the transition from scheduling unit S_1 to scheduling unit S_2 without a barrier. S_2 starts execution as soon as the branch condition at the end of S_1 becomes available[2]. Without a barrier, a static-dynamic dependence across the two scheduling units may not be respected, as the access commits may be re-ordered.

[1] A control-flow graph [41] of a program is a directed graph whose nodes correspond to basic blocks in the program, and edges correspond to branches between basic blocks.

[2] The condition for the branch at the end of S_1 is computed on some one tile, and then broadcast to all the tiles. The broadcast is absent if the end of the scheduling unit does not have a branch, or if it is unconditional.

Figure 5.3(b) shows the same two scheduling units with a barrier between them. S_2 does not start on any node until S_1 on every node is complete.

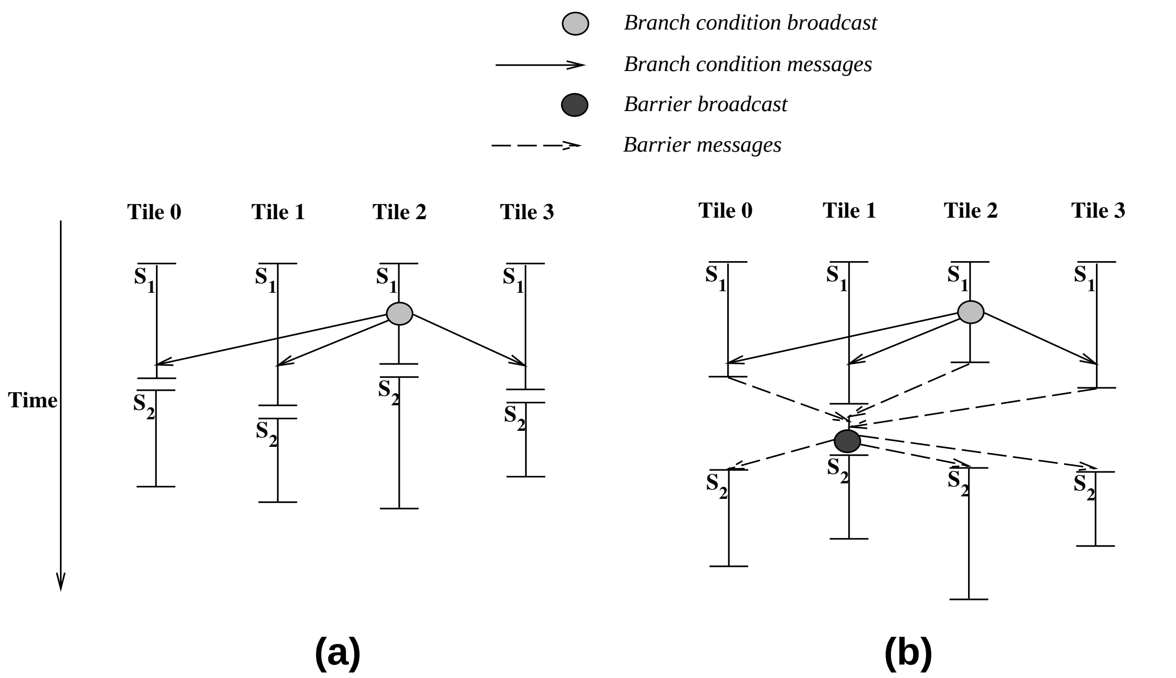


Figure 5.3: Transition between scheduling units without and with a barrier. In both (a) and (b), a 4-tile machine is assumed. S_1 is the predecessor scheduling unit; S_2 is the successor. Both S_1 and S_2 have been parallelized into 4 threads by Maps and the space-time scheduler. The vertical axis is time. The sets of 4 side-by-side vertical lines represent parallel threads of S_1 (above), and of S_2 (below). The branch condition at the end of S_1 is computed on a single tile (decided by the space-time scheduler; here tile 2) and broadcast to the other tiles. (a) Transition without a barrier. (b) Transition with a barrier in-between S_1 and S_2. The barrier starts with all tiles signaling completion of S_1 to a central tile (here tile 1). After all completion messages are received, the central tile broadcasts to all tiles signaling them to proceed to S_2.

Optimizing away memory barriers

There is a special case when a barrier is not needed. If in the predecessor, the last access for each equivalence class is of the same kind (static or dynamic) as the first access for that class in the successor, then no barrier is needed. The compiler should check for this condition. To see the utility of this check, consider *three common cases* in which the check automatically reveals that no barrier is needed. First, if all the accesses in both scheduling units are static, no barrier is needed. Second, if both scheduling units have only dynamic accesses, and all dynamic accesses are implemented through software-serial ordering, no barrier is needed. Third, if the predecessor and successor scheduling units are the same (*i.e.*, the scheduling unit is a loop body), and if each equivalence class in that scheduling unit has only one kind of access (static or dynamic), then too, no barrier is needed.

5.6 Dynamic optimizations

This section describes two optimizations for dynamic references that are handled by software serial ordering (SSO). The first, independent epochs, improves upon SSO by eliminating the turnstile's serialization in certain scheduling units. The second, updates, increases the number of cases when independent epochs are applicable.

Independent epochs

Isolating the memory references in different scheduling units, as in section 5.5, opens up the possibility of using different dependence strategies in different scheduling units. In particular, more aggressive schemes than software-serial ordering are possible for certain scheduling units with special properties, in case a distributed, software-controlled dynamic network is used. The optimizations are applied orthogonally for each equivalence class. We describe one of these optimizations, termed the independent epoch optimization, below.

The independent epoch optimization applies to code regions in which the compiler can determine that all accesses within a single equivalence class are independent from each other. We call such a region an *independent epoch* for that class. Such code regions are not related to the scheduling units used by the space-time scheduling compiler phase; independent epochs may be smaller than, equal to, or larger than the scheduling units. Normally in software serial ordering, all dynamic memory requests in a single alias equivalence class have to go through a turnstile for the entire duration of the program. Inside an independent epoch, all accesses issue in parallel, without going through a turnstile. Parallel issue is correct because the accesses are independent. Further, proper serialization with memory accesses outside the independent epoch is guaranteed by placing memory barriers before and after the independent epoch. Memory barriers are described in section 5.5. Memory barriers are needed before and after the independent epoch in all cases, regardless of whether barriers are needed if the same scheduling unit were implemented using software-serial ordering. A trivial example of an independent epoch is a region whose dynamic accesses to an equivalence class are all loads. Otherwise independent accesses are found using dependence analysis [26]; dependence analysis is aided by pointer analysis, array-index analysis and dataflow analysis.

Figure 5.4 shows an example of an independent epoch. Figure 5.4(a) shows the initial code with an $A[i+x]$ reference inside a loop. Assume that the variable x is not a loop index variable and does not have a known constant value; hence the $A[i+x]$ access is not affine. Further assume that the array $A[]$ is distributed using low-order interleaving. The $A[]$ reference is non-disambiguated because it is a non-affine reference to a distributed array. Figure 5.4(b) shows the loop after unrolling by a factor of 4, and application of independent epochs. The loop is unrolled to expose memory parallelism in the $A[]$ access. Dependence analysis shows that the 4 accesses after unrolling are independent because they are separated by constant offsets. Consequently, the 4 references form an independent epoch and are issued in parallel without a turnstile. Parallel dynamic access are shown in figure 5.4(b), and after code generation, in figure 5.4(c). Memory barriers before and after the independent epoch (not shown) ensure correct ordering with regard to references outside it.

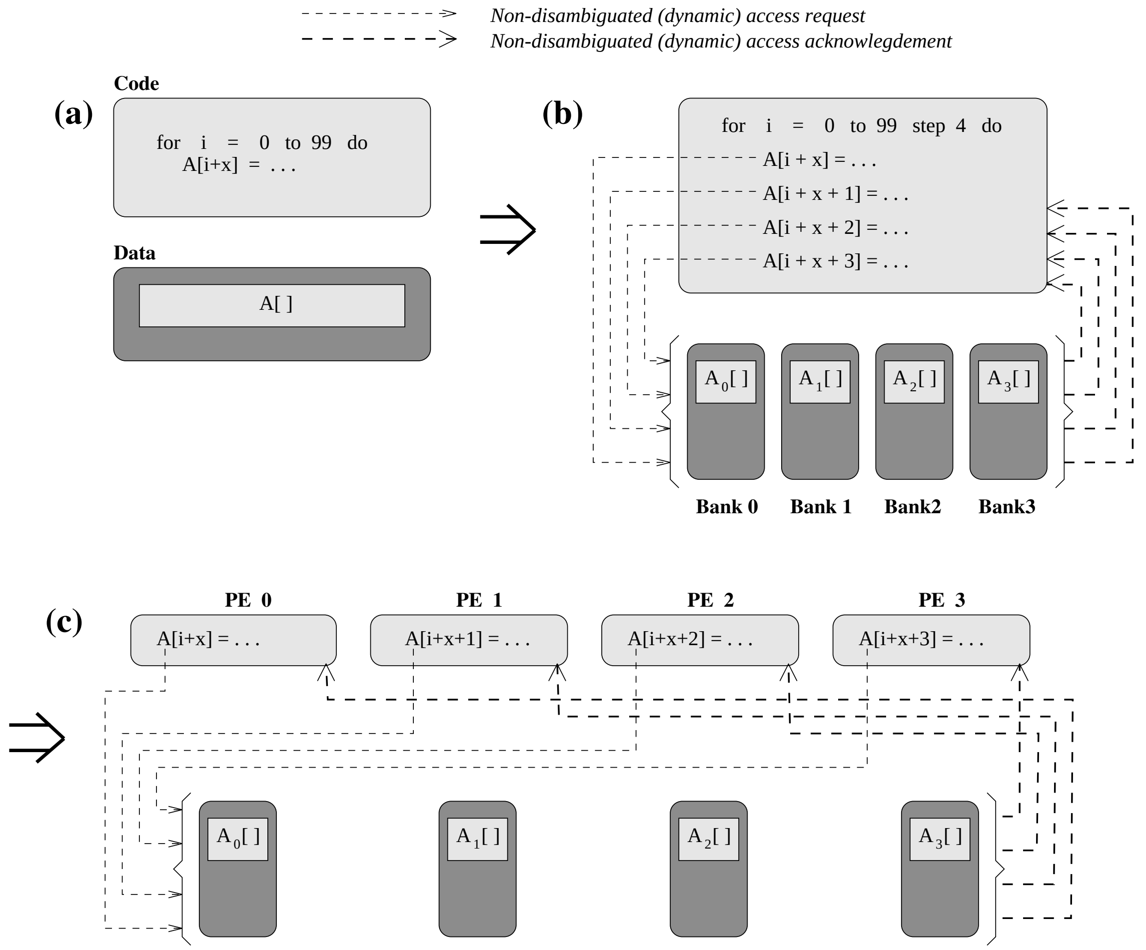


Figure 5.4: Example showing an independent epoch. (a) Initial code. The $A[]$ access non-affine. (b) After unrolling, dependence analysis and the independent epoch optimization. Dependence analysis finds that the 4 $A[]$ accesses are independent, and hence they are issued as an independent epoch, without serialization. (c) After code generation.

Updates

To understand the update optimization, consider the example in figure 5.5(a). Two read/modify/write operations on different elements of array $A[]$ are shown. As the values of the $B[]$ array are assumed unknown, the two $A[\cdots]$ locations alias in the case $B[i] = B[i+1]$; hence the two stores to $A[\cdots]$ are potentially dependent and must be serialized. The dependence between the two stores is shown by the dependence edge between them.

Figure 5.5: Example code benefiting from updates. (a) Initial code. Read/modify/write operations are performed on the two $A[\cdots]$ locations. The two stores to $A[\cdots]$ are potentially dependent and must be serialized. (b) Code after the update optimization. The two read/modify/write operations are moved into atomic message handlers for dynamic messages. Consequently, the two effectively have no dependence, since addition is associative and commutative. Hence the two read/modify/writes can be run as an independent epoch, as in described earlier in this section.

It is possible to improve the performance of the code in figure 5.5(a) in the following manner. Observe that if the two read/modify/write operations are each performed atomically, then the two operations are independent, since addition is both associative and commutative. The update optimization aims to benefit from independence by implementing the read/modify/write atomically. In particular, updates are memory handlers that implement simple read/modify/write operations on memory elements. For each read/modify/write operation, a dynamic message is generated that is sent to the memory bank containing the data operated upon. In this case, each $A[\cdots] + = \cdots$ operation is converted to a dynamic message. The actual read/modify/write operation is performed in the dynamic message handler. Handler atomicity guarantees update atomicity. Updates are possible on any software-exposed architecture whose dynamic network supports programmable active messages, such as Raw.

Figure 5.5(b) shows the code after the update optimization. Implementing the two read/modify/writes as atomic updates eliminates the dependence between them. Updates improve performance in two ways. First, an update collapses two expensive and serial dynamic memory operations, a load and a store, into one. Second, the associativity and commutativity of the updates effectively removes dependences between different updates. Dependence elimination helps increase the utility of independent epochs by finding regions with independent updates to an alias equivalence class.

The compiler migrates many different simple read/modify/write memory operations from the main program to the memory handlers. The modify operation is required to be both associative and commutative. Common examples include increment/decrement, add, multiply, and max/min.

In case floating point stability is required, updates cannot be applied to floating point addition and multiplication. Floating point addition and multiplication are not truly associative. For such operations, using updates retains correctness but not numerical stability. A compiler flag is provided that disables updates for floating point addition and multiplication upon user request.

Updates are a special case of the general technique of *moving computation to memory*. Moving computation to memory, possible on any software-exposed architecture whose dynamic network supports programmable active messages, refers to the process of migrating a code fragment normally in the main program into an active message handler. Moving computation to memory may be profitable for dynamic memory accesses if they refer to the same location multiple times, with computation between successive accesses. When profitable, the dynamic accesses with associated computation are packaged in a single active-message handler; the handler moves computation local to memory and obviates the need for long-latency communication between successive accesses.

5.7 Future work

While software-serial ordering has been fully implemented in the Raw compiler, the independent epoch and update optimizations have not. The evaluation of independent epochs and updates in the results chapter (chapter 9) is done using a hand-coded implementation. Future work will automate the implementation, and obtain more extensive results. The cost for memory barriers will be measured, and sophisticated strategies for when to use dynamic accesses will be investigated.

5.8 Summary

A summary of this chapter follows. The chapter begins by showing situations where dynamic accesses are useful. Efficiently implementing dynamic accesses is closely tied to enforcing dependence: schemes for implementing dynamic accesses aim to overlap as much of the dynamic latencies as possible while respecting all dependences. We show why dependences between dynamic accesses are hard to enforce efficiently. Software serial ordering is presented as a scheme to efficiently enforce dependence; it uses a combination of static and dynamic messages to enforce dynamic-dynamic dependences. Software serial ordering involves serializing the non-blocking requests to memory at a node called the turnstile; only references within each equivalence class, computed by equivalence-class unification, need to be serialized. Different equivalence classes use different turnstile nodes. Enforcing dependences across scheduling units is done by using memory barriers;

in some cases the barriers can be optimized away. Optimizing away the turnstile serialization is sometimes possible using the independent epoch optimization. The effectiveness of independent epochs is increased in some cases by the update optimization.

Chapter 6

Maps implementation

The next three chapters discuss implementing Maps as part of a complete compiler system. Several complex challenges arose in the implementation of the compiler for Raw. There is little in the literature on how to compile for bank-exposed architectures, so we were forced to devise our own strategies in many situations. The next three chapters document our solutions. In some cases, our solutions are not the only ones possible; nevertheless, the chapters document the issues that need to be considered, and provide one set of workable and consistent transformations and analyses. The next three chapters should be valuable to anyone wanting a deeper understanding of compilation for software-exposed machines, or someone involved in implementing such a compiler.

The next three chapters, chapters 6, 7 and 8, are organized as follows. This chapter describes the platform used and task flow of the Rawcc compiler. The software platform used in discussed in section 6.1. A detailed compiler flow, taking up most of the chapter, is presented in section 6.2. Each task in the compiler is explained – some tasks that require more involved explanations are explained in the following two chapters. Chapter 7 discusses issues relating to memory allocation and address management. Chapter 8 discusses the implementation of certain language-specific and environment-specific features.

6.1 Platform used

This section describes the software platform used for the Rawcc infrastructure. Rawcc is implemented using the SUIF version 1.2 compiler infrastructure [24]. SUIF provides a language-independent intermediate format for input programs, and a set of tools that implement traditional compiler tasks common to most architectures. SUIF also provides a set of tools to parallelize code for multiprocessors; however, we use none of the SUIF parallelization tools. The only SUIF tools used by Rawcc are related to traditional single-processor compiler tasks – all ILP and memory parallelization tasks had to be written in-house, customized for software-exposed architectures, rather than multiprocessors. Consequently, 18 out of 23 tasks in figure 6.1 were written in-house. For the remaining 5, we were able to use standard SUIF tools, or tools written elsewhere; for these 5 tasks, the tools used are mentioned in the task descriptions in this section.

6.2 Detailed compiler flow and description

This section lists out the tasks of the Rawcc compiler. Each task is discussed; many are illustrated through running examples.

Figure 6.1 shows the flow of the Rawcc compiler in detail. Only the tasks relevant to compiling for a software-exposed architecture are shown; traditional compiler tasks such as parsing, traditional code optimizations and back-end code generation are not shown. Each task in the figure is described in this section in the order of execution. Traditional code optimizations performed in the Raw compiler include constant propagation, loop invariant code motion, strength reduction and dead-code elimination; these are performed both before and after Maps.

Each of the compiler tasks in figure 6.1 is described below, in the order the tasks are performed.

Dismantle aggregate-object copies This task dismantles copies of aggregate objects, such as structures and arrays, into element-by-element copies. Certain programming languages allow aggregate objects to be copied to other aggregate objects of the same type by using the assignment operator. For example, ANSI C allows copying of entire structures in a single statement, but not entire arrays. Normally, the compiler's back-end dismantles each aggregate copy into an element-by-element or word-by-word copy. For Maps, however, all individual load/stores must be visible to the bank-disambiguation methods so that each load/store may be placed on the tile where its data resides. Hence, aggregate copies are dismantled before bank disambiguation, instead of the in the back-end.

The dismantling of each aggregate copy in this task produces an element-by-element copy, not a word-by-word copy. A word-by-word copy results in type conversions and un-typed copies; type conversions and un-typed copies violate our type-conversion restrictions, stated in section 7.1.

Build structured control-flow from unstructured This task builds structured control-flow from unstructured control-flow. The standard SUIF pass *porky* is used for this task with the flags for control-simplification and unstructured-control-flow optimization set. Modulo unrolling benefits from **for** loops, an example of structured control flow. This compiler task builds **while** loops, not **for** loops, from unstructured-control flow; **for** loops are built in the next task.

Convert while loops to for loops Performs induction-variable recognition, followed by code transformation, to convert **while** loops to **for** loops wherever possible. Modulo Unrolling benefits from **for** loops. The standard SUIF pass *porky* is used for this task with the flag for finding and building **for** loops set.

Forward-propagate to array indices Forward-propagation is a standard compiler task that moves expressions used in the calculation of local variables into uses of those

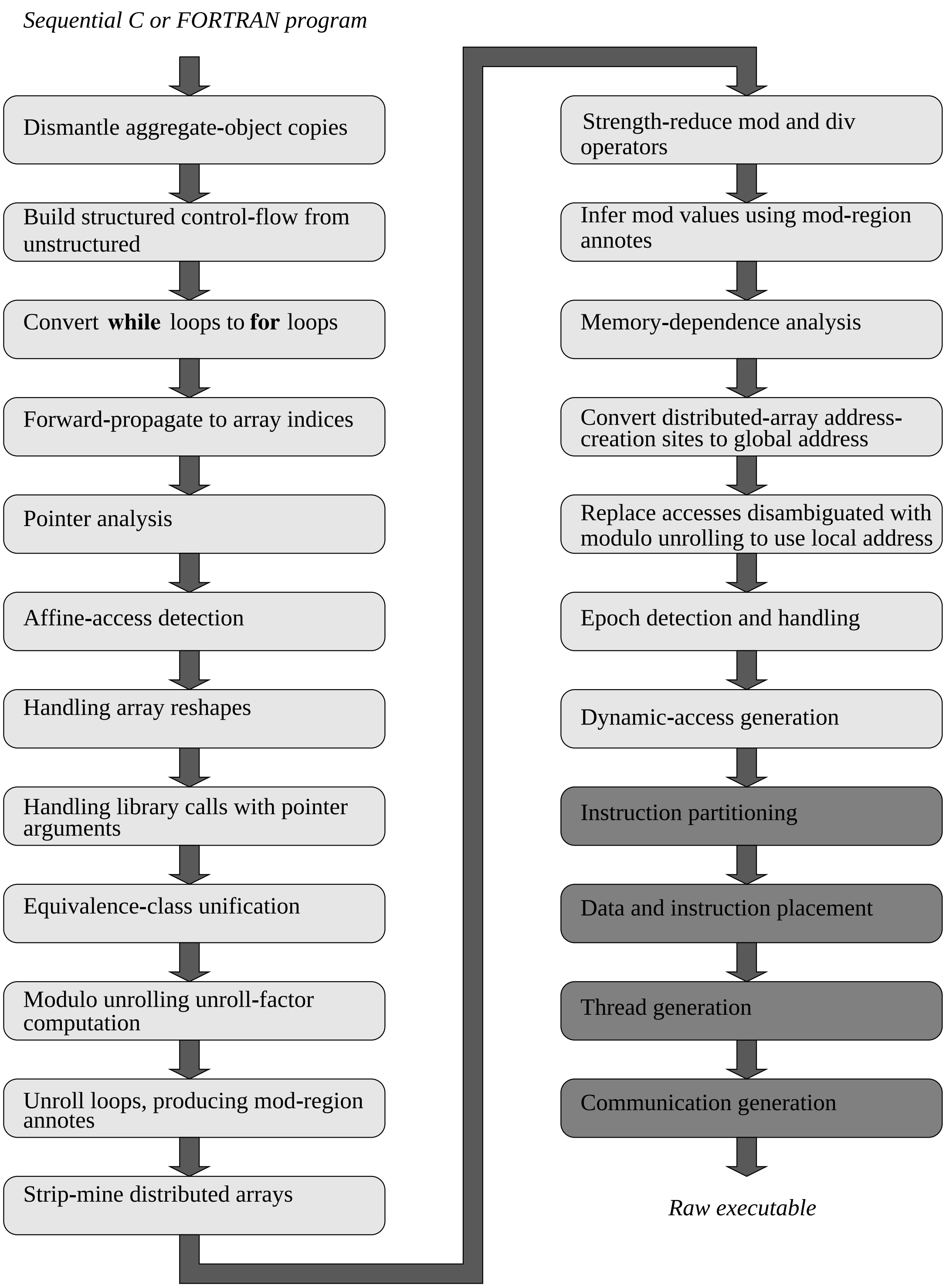


Figure 6.1: Detailed Rawcc compiler flow. Maps tasks are lightly shaded, space-time scheduler tasks (following Maps) are shaded darker.

variables when possible. Forward propagation applied everywhere is not desirable, as it performs the opposite of common sub-expression elimination, and may reduce performance.

There is one case where forward-propagation is helpful: when the local variables being replaced are part of the index expressions of array references. Such forward-propagation to array indices helps identify affine references; affine references perform well using modulo unrolling. Figure 6.2 shows an example of how forward-propagation to array indices helps identify affine references. Figure 6.2(a) shows a code fragment with an array reference. Figure 6.2(b) shows the same code fragment with forward-propagation to array indices run. In figure 6.2(b), the affine references have subscripts which are directly affine, aiding their identification as affine references.

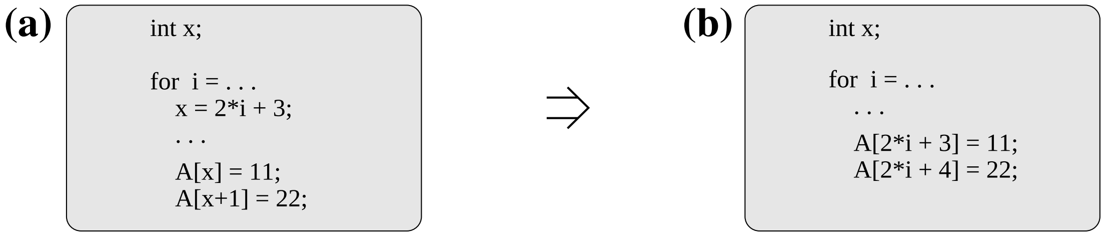


Figure 6.2: Example showing forward-propagation to array indices. (a) Initial code. (b) Code after forward-propagation to array indices.

Pointer analysis Pointer analysis is a compiler technique [35, 36] that identifies the possible data objects referenced by every memory-access instruction in the input program. Data objects are identified by their declaration or allocation sites. Maps uses SPAN, a state-of-the-art pointer analysis package [35], developed by Martin Rinard and Radu Rugina at MIT. Pointer analysis is used in Maps for three purposes: minimization of dependence edges, equivalence-class unification, and software serial ordering. Pointer analysis and its uses are discussed in detail in section 3.1.

Affine-access detection This task identifies array accesses to be affine whenever possible. Array accesses whose indices are affine functions of enclosing loop-induction variables are bank disambiguated using modulo unrolling. Such affine array-accesses need to be recognized in order for modulo unrolling to be effective. An affine-access detection library is implemented for Maps. The library performs algebraic simplification of expressions by applying the associative rules for multiplication over addition and subtraction. For example, the affine-access detection library simplifies $(3i - 4) * 2 + 1$ to $6i - 7$.

Handling array reshapes Array reshapes in FORTRAN allow arrays to be viewed with different dimensionality in different regions of the code. Section 8.3 discusses array reshapes in detail, along with how they impact bank-disambiguation. In summary, array reshapes disallow the padding optimization for distributed arrays in some cases. Maps handles such cases by not distributing the arrays, and disambiguating them using equivalence-class unification instead of modulo unrolling. See section 8.3 for details.

Handling library calls with pointer arguments Library functions in the Rawcc compiler are handled by placing each function entirely on one node. Placing the functions on one node allows the functions to be pre-compiled, instead of them being recompiled as a part of every program. Further, library calls that interface with I/O may need to be placed on one node for hardware reasons.

This task handles library functions that have pointer arguments. Section 8.2.2 describes this task in detail; a summary follows. Library functions that have pointer arguments access user-program data objects; hence the objects must be available on the tile where the library call is made. Consequently, Maps handles pointer arguments by forcing the different location-sets referred to by each pointer argument on to the same equivalence class. This way, since each equivalence class is mapped to a single tile by equivalence-class unification, each pointer argument refers to data on only one tile. Further, if there is more than one pointer argument, the different location-sets for different pointer arguments are all forced on to the same tile as well. Finally, the library call is placed on the tile where its pointed-to data resides. Different calls to the same functions can reside on different tiles.

Equivalence-class unification Equivalence-class unification (ECU) is described at length in chapter 3. A summary follows. First, the results on pointer analysis are used to construct a bi-partite graph. Next, connected components of the graph are found; the connected components form the equivalence classes. Finally, each equivalence class is mapped to a unique virtual bank. A virtual-to-physical mapping is done later in the compiler flow, in the space-time scheduler. ECU respects equivalence classes formed by forcing different location sets together into one equivalence class by the earlier pass of handling library calls with pointer arguments.

Modulo unrolling unroll-factor computation This task computes the unroll factor for each loop containing affine-function array accesses, using the method in section 4.2.

Unroll loops producing mod-region assertions This task unrolls each loop by the factor computed by the immediately preceding unroll-factor computation task. Unrolling in Maps differs from standard unrolling in that a pre-conditioning loop is generated if the lower bound of the original loop is not a compile-time known constant. The pre-conditioning loop guarantees that the main unrolled loop has a lower-bound value whose mod with the unroll factor is compile-time known. The known lower-bound modulo value is output in the mod-region compiler-inserted assertion. A lower bound for the main loop with known modulo value allows affine accesses in the main loop to go to a known bank using modulo unrolling.

Figure 6.3 demonstrates how Maps performs unrolling. Figure 6.3(a) shows a code fragment with an unknown lower bound. Figure 6.3(b) shows the code after unrolling by a factor of 4. The pre-conditioning loop ensures that the main loop starts at the next-higher multiple of the unroll-factor beyond the original lower bound x, in this case, $(x \text{ div } 4) * 4 + 4$. A mod-region compiler assertion is inserted on the main loop; the argument of the assertion is the condition that is guaranteed to be true by the compiler in the loop body. Here, the condition asserted is that the loop induction variable has a known mod value with the unroll factor in the main loop body. Hence, a mod-region assertion is of the form $induction - var \bmod unroll - factor = constant$; in figure 6.3(b), this is $i\%4 = 0$; the mod operator is shown as % in all code samples. The assertion is implementation is implemented as an annotation on **for** loops in SUIF. The mod-region annotation is used in the 'infer mod values using mod-region' task, later in the compiler flow.

(a)

```
for  i = x  to  99  do
    A[i] = i
```

$\Rightarrow$

(b)

```
int new_lb;

if (x % 4 == 0)
  new_lb = x
else
  new_lb = (x div 4)*4 + 4

for  i = x  to (new_lb-1)  do
    A[i] = i

for  i = new_lb to 99  step 4 do
//mod-region  i % 4 == 0
    A[i] = i
    A[i+1] = i+1
    A[i+2] = i+2
    A[i+3] = i+3
```

Figure 6.3: Example showing unrolling in Maps. (a) Initial code. (b) Loop unrolled by a factor of 4. The % operator is the mod operator. In (b), the lower bound of the main loop has a known value mod value with the unroll factor: here, $new\_lb \bmod 4 = 0$.

At this point, the extended example in figure 6.4 for $N = 4$ banks is introduced. The initial code in figure 6.4(a) involves affine accesses; hence the running example helps illustrate several of the subsequent tasks in the compiler flow that affect affine functions. Figure 6.4(b) shows the initial code after unrolling by a factor of 4, as required by a 4-banked memory system.

Strip-mine distributed arrays This task strip-mines arrays that Maps has decided to distribute (distributed arrays are low-order interleaved across the memory banks). Strip-mining an array with last-dimension size L replaces the last dimension with two

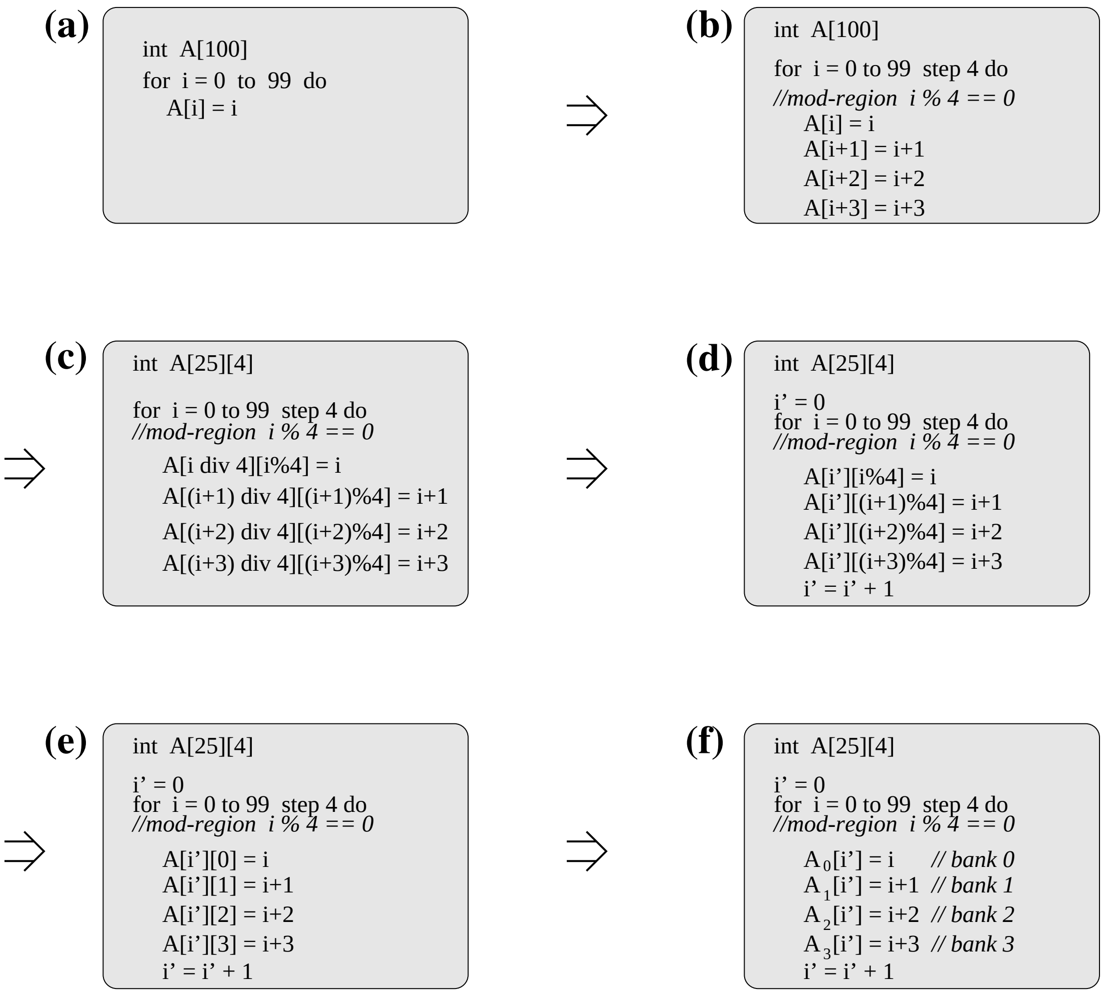


Figure 6.4: Example showing several tasks on code with affine accesses ($N = 4$). Italics are comments. (a) Initial code. (b) After unrolling, producing mod-region assertions. (c) After strip-mining. (d) After strength-reducing div operators. (e) After inferring mod values using mod-region assertions. (f) After replacing disambiguated accesses to use local addresses.

new dimensions of sizes $\lceil L/N \rceil$[1] and N, in that order. For example, strip-mining an array $A[8][102]$ for $N = 4$ banks changes the array to $A[8][26][4]$. Each array reference is correspondingly modified; for example, $A[i][j]$ is changed to $A[i][j\text{div}4][j \bmod 4]$. Strip-mining leads to the same result as when the last dimension of the array is padded to the next higher multiple of N, followed by strip-mining. Hence strip-mining conceptually performs padding; padding the last dimension is required for the padding optimization described in section 4.4.

Figure 6.4(c) shows the results of strip-mining array A for the code in figure 6.4(b) for $N = 4$ banks. $A[100]$ is replaced by $A[25][4]$, and the references are strip-mined such that, in general, $A[x]$ is replaced by $A[x\text{div}4][x \bmod 4]$; mod is shown as % in all code samples.

Strip-mining is useful for code generation: for any array reference after strip-mining, the new last dimension represents the bank number of the array element accessed, and the remaining dimensions represent the local offset of the local array on the bank. The compiler in later tasks aims to reduce the new last-dimension value for any array reference to a compile-time constant; if it succeeds, then the array reference is bank disambiguated to the bank number given by the constant.

At this point, the second of two extended examples is introduced: figure 6.5 illustrates several subsequent compiler tasks on a code fragment with dynamic accesses, for $N = 4$ banks. Figure 6.5(a) shows the initial code; figure 6.5(b) shows the code after strip-mining. The difference with figure 6.4 is that the array accesses are non-affine: x, y are variables other than loop induction variables.

Strength-reduce mod and div operators Integer remainder (mod) and integer division (div) operations are introduced into the code by strip-mining, the immediately preceding compiler task. Since mod and div are expensive operations at runtime, this task tries to optimize them away. A new strength-reduction technique [38] is used to optimize away div and mod operations whenever possible. Figure 6.4(d) shows the result of applying strength-reduction on the code in figure 6.4(c). The div operations are optimized away by introduction of a new variable i'; i' is incremented by 1 every time i is incremented by 4. The mod operations are not optimized away in this case.

Infer mod values using mod-region assertions This task simplifies down to constants, the mod operations introduced by strip-mining. The simplification is done with the help of the mod-region assertions introduced by unrolling, 3 tasks before this one. The mod-region assertions, applied to loop bodies, always have the format *var%int1 == int2*, where % is the mod operator. In this case, the mod-region assertion is $i \bmod 4 = 0$. Using the information that $i \bmod 4 = 0$, this task simplifies the mod values as follows: $i\%4$ simplifies to 0, and the rest are simplified using the associative rule for mod over addition and subtraction: $(a + b) \bmod c = (a \bmod c + b \bmod c) \bmod c$. Using this rule, all 4 mod expressions reduce to constants.

[1] $\lceil x \rceil$ denotes the ceiling of the floating point number x.

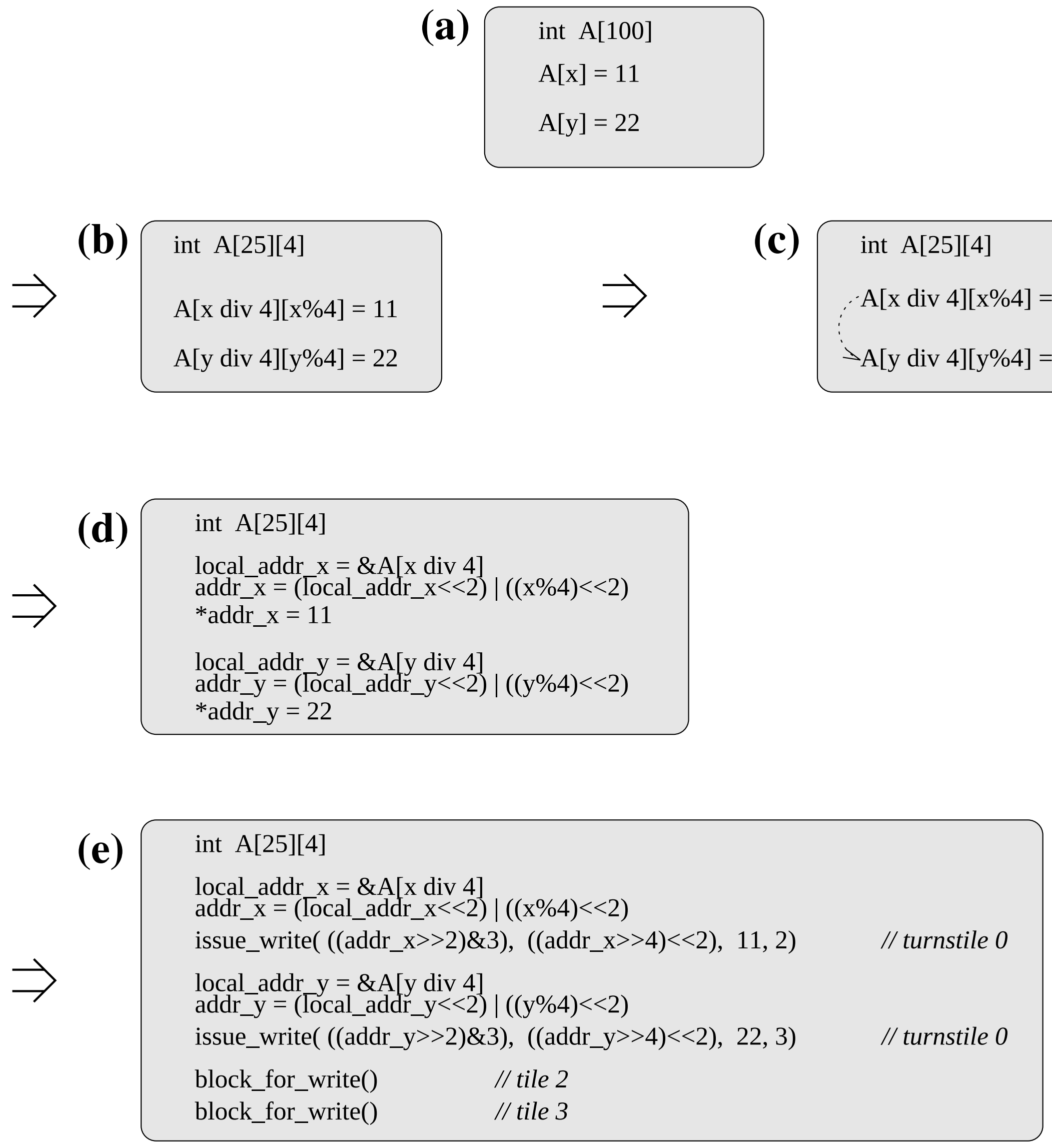


Figure 6.5: Example showing several tasks on code with dynamic accesses ($N = 4$). Italics are comments; % is the mod operator. (a) Initial code. (b) After strip-mining. (c) After memory-dependence analysis, dotted line is dependence edge. (d) After converting to global addresses. (e) After dynamic-access generation. *issue_write* is a dynamic store-request with parameters *issue_write*(*tile*, *offset_in_tile*, *value*, *return_tile*). The two requests are serialized on turnstile 0; the *block_for_write*() waits for the store-acknowledgments on return tiles 2 and 3. The particular tile numbers shown are virtual tile numbers. Their choice is incidental; they are re-mapped to physical tile numbers in the space-time scheduler.

Memory-dependence analysis This task introduces dependence edges between all pairs of memory references that can access the same location at run-time. Dependence edges are computed using pointer analysis, and further refined using array-index analysis. Section 3.1 describes how Maps computes dependence edges. Dependence edges are enforced in different ways, depending upon the nature of the dependent references. Static-static dependences are enforced by serializing on the disambiguated tile; static-dynamic dependences are enforced by explicit synchronization; dynamic-dynamic dependences are enforced by software serial ordering. See section 5.2 for details on enforcing memory dependences.

Figure 6.5(c) shows the results of memory dependence analysis on the code in figure 6.5(b). A dependence edge (dotted line) is introduced between the two array accesses, as they alias to the same location in the case x equals y. In figure 6.5(e), memory dependence analysis reveals that the four references are independent; hence no dependence edges are introduced.

Convert distributed-array address-creation sites to global address Representation of addresses is a complex issue in bank-exposed architectures: section 7.3 explains address representation in full. This task and the next involve address representation; both are summarized here. This task converts all address-creation sites for distributed arrays to generate global addresses[2]. Nothing needs to be done for address-creation sites not involving distributed arrays. The global address at distributed-array address-creation sites is computed in terms of the local address; the local address is first computed into a new compiler-introduced variable.

More precisely, for distributed arrays, the existing address is converted to a global address as follows. The array's last dimension is removed; the remaining expression is assigned to a local address. For example, for the strip-mined distributed-array address $\&A[i][j]$ creation site, the local address is $\&A[i]$, leading to the assignment $local\_addr = \&A[i]$. Then at the address-creation site, $\&A[i][j]$ is replaced by a global address variable assigned to $(local\_addr << \log(N))|(j << L)$, where N is the number of memory banks, L is the log of the array element-size (in bytes), $|$ is bitwise-or, $<<$ is left-shift, and $>>$ is right-shift. To understand this formula, see the format for global addresses in Maps shown in figure 7.1.

Figure 6.5(d) shows the code in figure 6.5(c) after the two distributed-array address-creation sites are converted to global addresses. The local and global addresses are derived using the formulas in the previous paragraph. This task is not shown for figure 6.4 since for affine functions, the next compiler task, 'replace accesses disambiguated with modulo unrolling to use local address', reverses the effect of this task.

---

[2]An address-creation site is a address expression in the program that is not simply an offset from an existing address, but an expression that computes an address of a data object using the 'address-of' operator ('&' in C), or an address returned from a heap-allocation routine. For example, a is an integer and p is a pointer, then $\&a$ is an address-creation site, but $(p + 1)$ is not. Address creation points for arrays are compiler operators for taking the address of an array symbol, and memory allocation call-sites used as arrays.

Replace accesses disambiguated with modulo unrolling to use local address This task replaces the addresses used in array references disambiguated by modulo unrolling to use local addresses. The local addresses are obtained from the corresponding array address-computation by dropping the last dimension. Since the access is disambiguated, its last dimension value, which gives the bank number, must be a constant. The constant bank number is placed as an annotation on the reference for use in tile assignment by the space-time scheduler.

Figure 6.4(f) shows the result of replacing the addresses of the memory references with local addresses, for the 4 array accesses disambiguated using modulo unrolling. The last dimension for each reference determines the particular local array accessed; a last dimension constant value c implies that the local array A_c is accessed. The bank number is also shown as a comment on the code. Figure 6.5 has no references disambiguated using modulo unrolling; hence this task has no effect on it.

Epoch detection and handling In this task, independent epochs are detected and handled. For details, refer to section 5.6. Independent epochs are not yet implemented in Rawcc, but an outline of a possible implementation is given in section 5.6. In summary, the implementation involves three steps. First, epochs are detected using a combination of pointer analysis, array dependence analysis, and relative memory disambiguation. Second, all dynamic accesses in the epoch are marked as having no turnstile; no turnstile assignment is therefore made for them by the subsequent dynamic-access generation task. Third, memory-barriers are placed immediately before and after the epoch.

Dynamic-access generation In this task, all non-disambiguated accesses are replaced with messages implementing a request-response model. Each load is replaced by an *issue_read*(*tile*, *offset_in_tile*, *return_tile*) call. Each store is replaced by an *issue_write*(*tile*, *offset_in_tile*, *value*, *return_tile*) call. The *tile* and *offset_in_tile* refer to the location of the remote address being accessed; the *return_tile* refers to the tile number where the reply bank should be sent. The *issue_read* and *issue_write* calls are inlined into dynamic-network message-launch instructions in the compiler back-end. In addition to an *issue_read*, each load also produces a $block\_for\_read()$ call. Similarly, in addition to an *issue_write*, each store also produces a $block\_for\_write()$ call. The $block\_for\_read()$ waits for a load-reply; the $block\_for\_write()$ waits for a store-acknowledgment.

Figure 6.5(e) shows the result of dynamic-access generation on the code in figure 6.5(d). The two stores, $*addr\_x = 11$ and $*addr\_y = 22$, are replaced by *issue_write* and $block\_for\_write()$ calls. Since the writes are memory-dependent, they are serialized through the same turnstile, in this case, mapped to tile 0. The two $block\_for\_write()$ calls are distributed to tiles 2 and 3 for parallelism. All tile numbers shown here are assumed to be *virtual tiles*; virtual tiles are an abstraction that assumes a potentially infinite number of tiles, and act as place-holders for physical tiles actually present on the machine. A many-to-one virtual-to-physical mapping is done later in the compiler in the 'data and instruction placement' phase of the space-time scheduler. The actual virtual tile values chosen, in this case 0, 1 and 2, is incidental; the virtual-to-physical mapping re-maps these numbers to physical bank numbers.

The space-time scheduler guarantees that there is at most one outstanding $block\_for\_...$ call of either type outstanding at one time. Having at most one $block\_for\_...$ call has the advantage that at run-time, time need not be wasted checking for which dynamic access an incoming reply corresponds to. If there are more $block\_for\_...$ calls than the number of tiles, then some tiles may have multiple $block\_for\_...$ calls. If a tile has multiple $block\_for\_...$ calls, the corresponding requests for the memory accesses are serialized to ensure that there is only one outstanding $block\_for\_...$ call at one time.

Space-time scheduler

The 19 tasks above implement Maps; the remaining 4 tasks in figure 6.1 compose the space-time scheduler [25]. At the conclusion of Maps, all memory objects and references are mapped to virtual tiles. The space-time scheduler partitions the instructions into multiple tiles and produces an executable. Maps assigns memory instructions to virtual tiles; the space-time scheduler assigns non-memory instructions to virtual tiles. Next, the space-time scheduler does a virtual-to-physical mapping of the tiles. Subsequently, multiple threads are generated, one per tile. Finally, communication is generated.

The space-time scheduler tasks are outlined below. For a more detailed description, see [25].

Instruction partitioning This task assigns non-memory instructions to virtual tiles. Tile assignment is done to optimize for locality, *i.e.*, instructions that sequentially depend on each other are favored to be placed on the same tile to avoid inter-tile communication. Mostly independent streams are placed on different tiles for parallelism. Some register-communication is inevitable – the assignment tries to balance the goals of minimizing communication and maximizing parallelism.

Data and instruction placement This task performs a virtual-to-physical mapping on tile numbers. The placement phase takes into account the target machine's register-level communication network's topology. The number of hops on the network determines the non-uniform access latency for any communication message. The Raw machine uses a 2-dimensional mesh topology. This task aims to minimize the number of hops traveled for most messages.

Thread generation This task splits the instructions and data into multiple threads, based upon the tile assignments made by the placer. Stacks and aggregate objects are padded, as required by Maps. Section 7.2 explains padding for stacks and aggregate objects.

Communication generation In this task, intermediate-node routing for register-level communication is performed. Such intermediate-node routing is required if the target architecture uses a compiler-routed network. Routing is done while ensuring that the routing schedule is deadlock-free. To minimize communication volume, multiple messages from the same source are serviced using a single multi-cast operation.

At the end of the space-time scheduler, the compiler back-end (not shown) generates machine-code instructions specific to the ISA of the target architecture. The Raw prototype uses a MIPS R4000 ISA for each tile, augmented with communication-access primitives, and bits for static-network routing. Rawcc uses the Machsuif [42] back-end code generator, modified to produce communication instructions and routing bits, in addition to standard MIPS instructions.

Chapter 7

Memory allocation and address management

This chapter discusses implementation issues regarding memory allocation, and the representation and management of addresses in Maps. There are two kinds of addresses in Maps: local and global. Disambiguated accesses use *local* addresses to access memory, since disambiguated accesses on Raw go to local banks. Non-disambiguated accesses refer to memory on a compile-time-unknown bank; hence non-disambiguated accesses use software-constructed *global* addresses. This chapter discusses issues related to maintaining two address representations. The chapter also describes how distributed stacks and aggregate objects are implemented.

This chapter is organized as follows. Section 7.1 describes the representation and handling of addresses in bank-exposed architectures. Section 7.2 shows how address management of aggregate objects is done by the compiler. The section also shows how stacks are distributed in an manner analogous to aggregate objects. Section 7.3 describes how the compiler keeps track of which addresses are local, and which are global. Section 7.4 discusses how the effectiveness of pointer analysis varies with different kinds of analyses used.

7.1 Address representation and handling

This section describes the format of addresses in Rawcc and their handling in the compiler. First, the format of both disambiguated and non-disambiguated addresses is described. Second, how these addresses are interfaced to the hardware is explained. Finally, a discussion of the choice of address format follows, including a discussion on type-dependent and type-independent formats.

Address format

This section discusses how Maps handles addresses for disambiguated and non-disambiguated accesses. Recall that Maps begins with the data objects in a sequential program and through its disambiguation methods, derives a precise data distribution for each object. This data can be referenced through either disambiguated or non-disambiguated accesses. For disambiguated accesses, by definition the particular cache bank they access is fixed and known; hence their address expressions are maintained in Maps as local within the bank accessed. The bank number is specified by annotating the access with the bank number, and ensuring that the later phases of the compiler schedule the reference to be local to that bank. In Rawcc , disambiguated references are always made local.

Address representation for non-disambiguated accesses is more complex. For such accesses, the bank accessed is unknown and may vary at runtime. Maps maintains their addresses as global in software. While many bit-encodings of global addresses are possible, Maps uses a low-order interleaved format, *i.e.*, the low-order bits represent the bank number, and the high-order bits represent the offset within the bank. The lowest-order bits represent a byte-offset, appended to allow pointer arithmetic. Figure 7.1 shows this representation of global addresses, and its relationship to local addresses. As an example, for a 32-bit Raw machine with 8 banks, a pointer to integer would have bit widths of 2, 3 and 27 for the byte offset, bank number, and offset-within-bank fields respectively. For any such format, pointer arithmetic in bytes continues to work within arrays or heap-allocated chunks of memory which have been low-order interleaved by modulo unrolling[1].

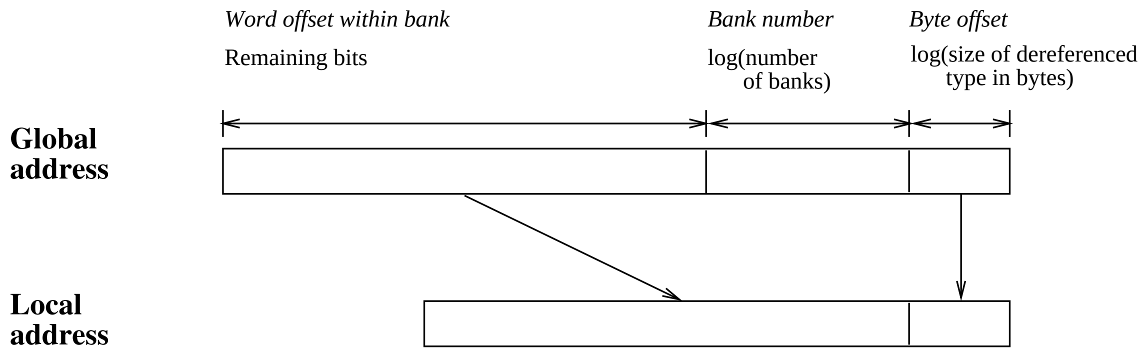


Figure 7.1: Address representation in Rawcc. Global addresses are shown using a low-order interleaved format. Local addresses within a tile correspond to the global address without the bank number. While global addresses are used for non-disambiguated references, local addresses are used for disambiguated references.

[1] To make this work the offsets of all such aggregate objects have to be the same on all memory banks. Section 7.2 describes how this is achieved.

Interfacing addresses to hardware

How the above software global addresses are interfaced to hardware depends on whether the software-exposed architecture provides a shared or distributed hardware view of memory. If a shared memory view is provided, global addresses are used as hardware addresses. If a distributed memory view is provided, Maps has the additional task of converting non-disambiguated global references into messages with message handlers using local addresses. For each such reference, the bank number and local address are extracted from the global address into separate words in software, and then used as the target and contents of remote-request messages respectively. Remote message handlers perform a local access on the address in the contents. This is the approach in the Raw machine. Section 2.4 describes how load/stores are converted to messages in Raw.

Type-dependent vs. type-independent address formats: type-conversion restrictions

Address formats used in Rawcc , as specified in figure 7.1, are type-dependent, *i.e.*, they use an address format whose bit-widths depend upon the type of the data pointed to. Our type-dependent format interleaves at the granularity of data elements, not any fixed number of bytes. For example, for an array of integers (4-byte elements), interleaving is at the 4-byte level; for an array of doubles (8-byte elements), interleaving is at the 8-byte level. To allow interleaving at a type-independent number of bytes, interleaving must be done at the granularity of the largest scalar elements; any smaller fixed granularity splits the largest scalar elements into more than one bank. Since single elements must not be divided to different banks, type-independent formats interleave at the granularity of the largest scalar elements; for most compilers, these are doubles.

The advantage of type-dependent formats is that they offer the most memory parallelism. Using a type-independent format results in a loss of memory parallelism. For example, for array $A[]$ of integers, $A[0]$ and $A[1]$ are mapped to the same bank in a type-independent format that interleaves at an 8-byte level. The disadvantage of type-dependent formats is that they do not allow the same array to be accessed as arrays of different types having different element sizes; thus programs with type-casts between arrays of different types are not allowed.

Since Rawcc uses a type-dependent format, it disallows programs with arrays cast to different types[2]. While Rawcc uses a type-dependent format, it is possible to build an implementation of Maps that uses a type-independent address format. To avoid the loss of memory parallelism when element-sizes are smaller than the largest element size, it is possible to use an optimization that uses a type-independent format for the object only when type conversions are possible on it.

[2]The SPAN pointer analysis package disallows programs with arrays cast to different types anyway, as described in section 7.4, so no additional restriction is placed because of Maps.

7.2 Aggregate objects and distributed stacks

This section describes the challenges in allocating and managing the addresses of aggregate objects that are distributed among multiple exposed memory banks. The solution used in Maps is outlined. The same solution is used to manage distributed stacks as well; distributed stacks are those that are distributed to multiple banks.

Representation of the addresses of distributed aggregate data objects such as arrays and structures presents a problem in a distributed memory system. In order to gain memory parallelism, the disambiguation methods in this thesis distribute these objects among multiple cache memory banks. Doing so may result in base offsets on different banks that may differ. This is highly problematic, as now every pointer to a distributed object has to be represented by a different pointer on each tile. Furthermore, pointer arithmetic on array-like objects does not directly work if the base offsets of an array are different on different banks.

A simple yet effective solution to this problem of unequal base addresses is to keep the base address for every aggregate object aligned across all the banks. Alignment is achieved by padding the object on each tile to the largest size on any tile. Padding ensures that distributed objects have the same address on each tile. Padding is done for all distributed objects, as well as for stack frames and heap-allocated objects. Padding in this manner has several advantages. First, a single address now suffices to represent all pointers, such as program pointers, the stack-frame pointer and pointers internal to heap-management routines. Second, address computation for any reference can be done on a different tile from the reference, increasing parallelism and scheduling flexibility. Finally, pointer arithmetic works within distributed array-like objects. For arrays, modulo unrolling distributes the arrays in a low-order interleaved manner, giving good load balance. Maps aims to distribute the scalar objects inside structures and stack frames to different banks as much as possible, so good load balance is likely.

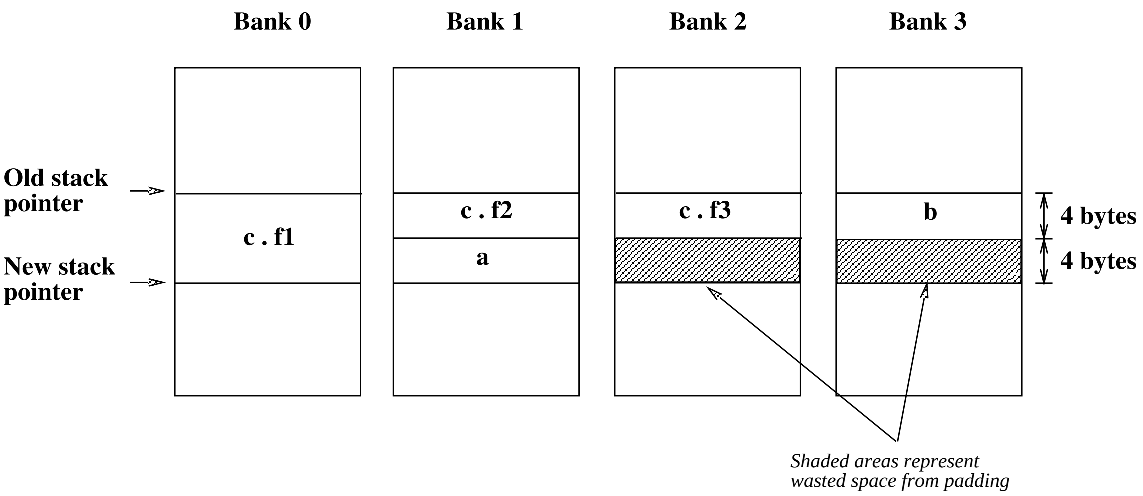
Distributed stacks are implemented in a manner similar to the implementation of aggregate objects. The stack elements are distributed among the tiles in a load-balanced manner, with the beginning of the stack frame identical across all the tiles. Alignment across tiles is achieved as before by using padding; in this case, padding the stack frame on each tile to the largest size on any tile. Thus, a single value represents the stack-frame pointer on all tiles, instead of having a separate pointer on each tile. For memory parallelism, it is desirable to distribute different elements in a stack frame, as is ensured by this scheme.

Figure 7.2 illustrates the implementation of both stacks and aggregate objects. Figure 7.2(a) shows a code fragment with data declarations including a structure aggregate object. Assume that the declarations are for the stack. Figure 7.2(b) shows the layout of the objects in memory when the stack is allocated using the scheme described in this section on a 4-banked system. The stack frame's beginning is identical on all the tiles, and so is the beginning of the structure. The stack frames are padded on banks 2 and 3 to maintain the alignment for subsequent stack frames.

```
int a;
int b;

struct {
  double f1;
  int f2;
  int f3;
} c;
```

(a)



(b)

Figure 7.2: Sample layout of aggregate objects and stacks. (a) Code fragment with data declarations for the stack. (b) Data layout of the objects on a 4-banked memory system for a software-exposed architecture. Doubles are assumed to take 8 bytes; integers, 4 bytes. The beginnings of both the stack frame and the structure object are kept aligned across the different banks. This invariant is maintained by padding at the end of the stack frame.

7.3 Global and local addresses

This section describes how Maps keeps tracks of which addresses in the program are global, and which are local. The objective is to ensure that each memory reference gets the address format it expects in its address.

While disambiguated references require local addresses and non-disambiguated references require global addresses, it is not easy to divide up all pointers used in a program at compile-time based upon their ultimate usage. Yet this distinction is important because at the usage points, namely references, the representation assumed for the reference address, *i.e.*, local or global, must be the actual representation of the address at that point in the program. It is difficult to classify all pointers as their ultimate usage may not even be known, and different uses may refer to the object in different ways.

The solution used by Maps is based upon the following observation: the only single location sets that could be distributed are arrays[3]. (Structure fields are assigned to different location sets.) Hence the only pointers that could refer to distributed location sets, and hence be used in non-disambiguated accesses, are those that could point to distributed arrays. Based upon this observation, Maps always constructs global addresses at address-creation points for distributed arrays[4], leaving other pointers local. A local-address expression is created first; the existing address-creation expression is replaced by one that constructs a global address in terms of the local address. The local address is also computed because it may be needed later. Address manipulation instructions are left unchanged. Finally, in array references disambiguated through modulo unrolling the global address is replaced by a local address.

The strategy in the previous paragraph ensures correctness using two properties. First, all possible non-disambiguated references get global addresses (the desired behavior) as they must be to distributed arrays. Second, references disambiguated using ECU get local addresses, as they must be to non-arrays. For array references disambiguated through modulo unrolling, the only remaining case, the global address is replaced by a local address. This replacement is done as follows. At each array reference disambiguated by modulo unrolling, the address used is replaced by the new local-address variable generated at the address-creation site. If the global address is never used, *i.e.*, when the address computation is used only for disambiguated accesses, then the global-address variable-assignment is dead-code eliminated by a subsequent dead-code elimination pass. The strategy of computing both global and local addresses at address-creation sites avoids having to keep track of all the uses of an address at its creation point.

[3] Or heap-allocated blocks used as arrays.

[4] An address-creation site is a address expression in the program that is not simply an offset from an existing address, but an expression that computes an address of a data object using the 'address-of' operator ('&' in C), or an address returned from a heap-allocation routine. For example, a is an integer and p is a pointer, then $\&a$ is an address-creation site, but $(p+1)$ is not. Address creation points for arrays are compiler operators for taking the address of an array symbol, and memory allocation call-sites used as arrays.

7.4 Effectiveness of pointer analysis

This section describes how pointer analysis may limit the class of programs accepted, based upon the kind of pointer analysis algorithm used. The trade-offs among the different kinds of algorithms possible are stated.

Pointer analysis packages are of two types: those that track conversions between pointers and integers [43], and those that do not [35, 36]. The first kind of package, *i.e.*, those that track conversions, ensure that pointer information is retained when a pointer is converted to an integer; some arithmetic done on the integer; and the result cast back to a pointer. The second kind of package, *i.e.*, those that do not keep track of conversions, lose information when a pointer is converted into an integer; integers are not tracked, and pointers cast back from integers are assumed to point to any address. The advantage of the second kind of package, however, is that they can afford to use more precise and time-consuming algorithms, since they track only pointer values, a smaller set that the set of all pointer and integer values.

Maps can use any pointer analysis package; however, the Rawcc implementation of Maps uses the SPAN package [35]. SPAN does not allow casts between pointers and integers; hence, neither does Rawcc.

Chapter 8

Language and environment features

This chapter describes how certain language-specific and environment-specific features are implemented in Maps. Features that have special requirements and constraints on the data they access need to be handled in Maps, as Maps allocates all the data accessed by the program. Features that require special handling in Maps are presented: the handling of procedures, library calls and array reshapes in FORTRAN are discussed. For example, a convention needs to be decided on the locations used for the parameters and return values of procedures. Further, the procedure stack needs to be distributed. Library calls also have special requirements on their input and output data. Array reshapes in FORTRAN restrict the data allocation strategies that can be used for bank disambiguation. Rawcc currently accepts C and FORTRAN programs. Though the features are discussed in the context of the languages that the Raw compiler Rawcc supports, they can be generalized to similar constructs in other languages.

This chapter is organized as follows. Section 8.1 describes how procedure calls are implemented in Maps. Section 8.2 discusses the issues involved in implementing library calls, and presents the approach taken in Raw. Section 8.3 describes how array reshapes in FORTRAN are handled.

8.1 Procedure Calls

This section is describes how procedures are handled in Maps. The focus on how their memory is handled, *i.e.*, memory for their input arguments, return values and local stack.

Maps classifies procedures into two types: parallel procedures, and sequential procedures. Parallel procedures are those procedures whose code is distributed over all the tiles; execution proceeds in parallel across the different tiles. Most procedures in Rawcc are, by default, made into parallel procedures. Parallel procedures enable the full exploitation of ILP and memory parallelism. A parallel procedure is implemented by creating a local procedure on each tile that implements a portion of the original sequential procedure in the input program. A single procedure call in the input program is translated to a procedure call on every tile in the executable; each tile calling its own local procedure. The original procedure's stack is implemented as a stack distributed across the local procedures, such that each original stack object is mapped to one of the stacks of the local procedures. Handling of distributed stacks is detailed in section 7.2.

Figure 8.1 demonstrates how procedures are made into parallel procedures. Figure 8.1(a) shows the initial code of a procedure *foo*(). Procedure *foo* takes in an integer argument x, and returns an integer value, besides allocating three local variables, a, b and c, and a local array $D[]$. At the end of the procedure, the value in integer b is returned. A single call to *foo*() is also shown. Figure 8.1(b) shows the code for a 4-tile machine after Rawcc has converted *foo*() to a parallel procedure. Procedure *foo* is cloned into 4 local procedures, foo_0, foo_1, foo_2, and foo_3, one on each tile. The space-time scheduler allocates the arguments and return values to particular tiles; in this case, argument x is placed on tile 1, and the return value is placed on tile 2. The stack variables a, b and c are distributed among the local procedures. The stack array $D[]$ is assumed distributed using modulo unrolling, resulting in 4 low-order interleaved local arrays being allocated, one on each tile. The body of the procedure (not shown) is partitioned across the tiles. The call to *foo* translates to a parallel call on all 4 tiles, with each tile calling its local version of *foo*(). Tile 1 passes in the argument x, tile 2 receives the return value.

Sequential procedures, on the other hand, are those that are executed entirely on one tile. Sequential procedures resemble procedures in most conventional microprocessors, in that their entire code and stack are mapped to one tile. Sequential procedures can be called only from the tile on which they reside, however, their parameters may be obtained from, and results send to, other tiles.

While parallel procedures provide the most parallelism, there are situations where Maps chooses to make procedures sequential. All library calls are made into sequential procedures. Section 8.2 describes why library calls are made sequential. In addition, sequential procedures may perform better than parallel procedure for very small procedures that have no parallelism and perform no memory accesses. If such small procedures are made parallel, only one tile might have any computation in it. It may be better to make the small procedure sequential to avoid the overhead of all the tiles making calls to local procedures, when only one tile has anything in its local procedure. Further, when small procedures are made sequential, different calls to small procedures can exploit parallelism between the calls. Such parallel calls to small procedures commonly arise when a loop with a small-procedure call in its body is unrolled.

8.2 Handling libraries

This section describes the special features needed to handle library calls in Maps. The reason for why they need special handling is described. Next, the method of using pointer analysis stubs is shown; pointer analysis stubs allow library calls to be pre-compiled without running pointer analysis on them. Finally, the handling of library calls with pointer arguments is described.

Library calls need special handling in a compiler for a bank-exposed architecture for two reasons: first, library functions may need to be sequentialized to a single tile, and

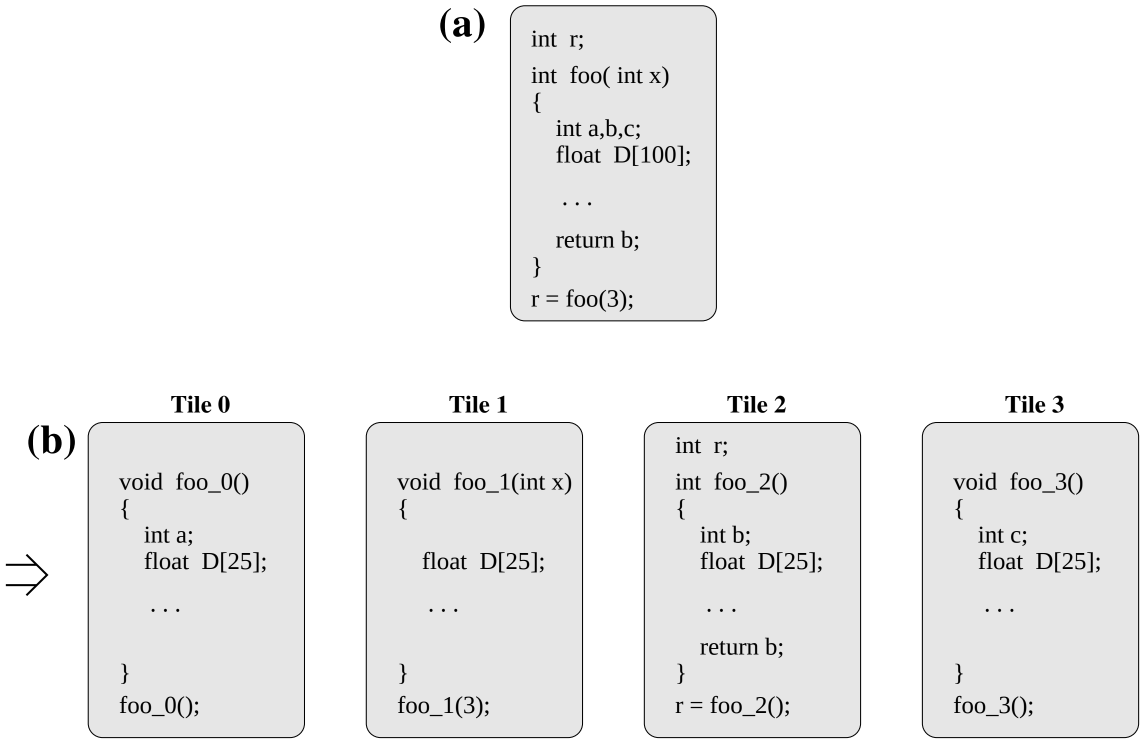


Figure 8.1: Example of a parallel procedure. (a) Initial code. (b) Code after procedure *foo*() is made into a parallel procedure on a 4-tile machine. Procedure *foo*() is cloned into 4 local procedures, and the call to *foo*() is also cloned. The stack data and the code is distributed among the 4 tiles.

second, pre-compilation of libraries may be desired. Each reason is discussed in turn. First, library functions may need to be sequentialized to a single tile, *i.e.*, handled as sequential procedures. Library calls dealing with off-chip input and output may need to be on a single tile if the I/O handling mechanism so demands. Guaranteeing correct sequential semantics is easier if the I/O routines are sequentialized to one tile. Some library functions may have complex code that violates the type-conversion restrictions of Rawcc , stated in section 7.1, and thus cannot be parallelized by Maps. The second reason for special handling of library calls is that it is desirable to pre-compile library calls once, rather than re-compile them for every application. A consequence of pre-compilation is that at the time of compilation of the libraries, the pointer analysis information of the applications are not known. Consequently, simplifying assumptions are made about the location of program data accessed by the library function; all program data accessed by a library function is assumed to reside on one tile, and the library function is scheduled on that single tile.

Having pre-compiled, sequential, library functions has two further implications on the implementation of library calls. Section 8.2.1 describes the first implication, *i.e.*, pointer analysis stubs. Section 8.2.2 describes the second implication, *i.e.*, handling of library calls with pointer arguments and return values.

8.2.1 Pointer analysis stubs

For library functions having arguments or return values that are pointers, Rawcc needs to know a summary of their pointer behavior at compile-time. For example, consider the template of the *strtol*() function in figure 8.2(a). *strtol*() returns as a long integer the value represented by the character string pointed to by *str*. The string is scanned up to the first character inconsistent with *base*. The address of the remaining string, after the number, is returned in the *ptr* parameter. For the *strtol*() function, Rawcc needs to know at compile-time that the location-set returned by the *ptr* parameter is the same location-set passed in through the *str* parameter. Without running pointer analysis on the library function, the location-set returned by *ptr* is not known. Running pointer analysis on the library functions is inconvenient, as the pre-compiled libraries would need to be re-compiled through pointer analysis. Re-compilation is time consuming, and may not work if the library function violates the type-conversion restrictions of Rawcc , stated in section 7.1.

Pointer analysis stubs are an innovative solution, first proposed by Robert Wilson in his thesis [44], that provide a summary of the pointer behavior of library functions, without the need for re-compiling them through pointer analysis. A pointer analysis stub for a library function is a short procedure having the same template as the library function, that summarizes the pointer analysis information of the library function. Pointer analysis stubs are *not* correct implementations of the function. They are linked in at application compile-time, and discarded after pointer analysis. Figure 8.2(b) shows the pointer analysis stub for the library function template in figure 8.2(a). The pointer analysis stub tells pointer analysis that the location-set pointed to by *ptr* is the same location-set (object)

(a)
```
// strtol() returns as a long integer the value
// represented by the character string pointed
// to by str.  str is scanned up to the first
// character inconsistent with base.  The
// remaining string is returned through the
// parameter ptr.

long  strtol(char *str, char **ptr, int base);
```

(b)
```
// Pointer analysis stub for strtol()

long strtol(char *str, char **ptr, int base)
{
    *ptr = str;
    return 1;        // return any bogus number
}
```

Figure 8.2: Example of a pointer analysis stub. (a) Template of the strtol() library function. Its functionality is explained in a comment. (b) Pointer analysis stub for the strtol(). The pointer analysis stub summarizes the pointer behavior of the function; it is not a correct implementation of strtol().

as that in *str*. Pointer analysis stubs are usually very simple, and rarely exceed a few lines of code. Pointer analysis recognizes the *malloc* function as a special case that returns a new location-set not aliased with any other object in the program; hence *malloc* has no stub. Malloc may be used in the pointer analysis stubs of other library calls to express a newly-allocated block of memory.

8.2.2 Library calls with pointer arguments

Since library functions are sequential, it is assumed that parameters and return values that are pointers point to data on a single tile. Consequently, Rawcc must ensure at application compile-time that all pointed-to data for each library call resides on a single tile. Single-tile data for each library call is ensured as follows. For each library call, the location-sets of all the pointed-to objects are forcibly merged into one equivalence class during equivalence-class unification. Therefore, since each equivalence class is mapped to a single tile by equivalence-class unification, all pointer arguments and return values refer to data on the same tile. Finally, the library call is placed on the tile where its pointed-to data resides. Different calls to the same library function may reside on different tiles, enabling parallelism between the calls.

8.3 Array reshapes in FORTRAN

This section describes what array reshapes in FORTRAN are, and why they need special handling in Maps. Array sections, which could occur in C as well, are also described, as they are handled in a manner similar to array reshapes.

Array reshapes in FORTRAN have implications on whether the padding optimization for modulo unrolling can be applied. An array is said to be reshaped if it is accessed in different parts of the program with a different number of dimensions, or different sizes of dimensions. For example, if an array A of 100 elements is accessed in the program assuming dimensionality $A[100]$, but accessed elsewhere assuming dimensionality $A[25][4]$, then the array A is said to be reshaped. Array reshapes can happen in FORTRAN in three ways: parameter reshapes, equivalences and different common block declarations. For details, see [39]. An example of a reshape is when an array is be passed as an argument to a procedure that accesses it with different dimensionality. In any reshape, the correspondence of the elements between different versions of the same array is one-to-one and in-order on the column-major ordering of the array.

Array reshapes have no effect on Maps if the padding optimization, described in section 4.4, is not applied. Without padding, the array elements are low-order interleaved based upon the total offset of the array element (in row-major or column-major ordering, as the case may be). By definition, the total offset of an array element does not change when the array is reshaped; hence, modulo unrolling produces the same bank number and offset for an element across different reshapes, without any special handling. Consequently, without padding, reshapes do not affect Maps.

With the padding optimization, however, array reshapes make the default handling of distributed arrays incorrect. The padding optimization, applied on distributed arrays, low-order interleaves the array based upon the last dimension value alone. Across different reshapes, using the last dimension value for finding the bank number and the remaining dimensions for finding the offset within the bank is incorrect, since for different reshapes, different bank numbers and offsets incorrectly result for the same array element.

The current implementation of Maps uniformly applies the padding optimization to all arrays, and uses the last dimension value for low-order interleaving. Consequently, as padding makes reshapes incorrect, any array that is padded in any of its reshapes is not distributed – it is placed entirely on one node and handled using equivalence-class unification. Padding is always applied since not padding distributed arrays either increases the code size and code generation complexity in case disambiguation is desired, or makes all the array references dynamic. Neither scenario for not padding is very attractive, and have not been studied in detail in the Raw compiler.

Passing array sections

A similar concept to array reshapes is the passing of array sections. An array section is said to be passed when the actual argument for an array parameter is a pointer to the middle of an array, rather than its beginning. This makes modulo unrolling, used for distributed arrays, incorrect for the following reason. The code generation phase of modulo unrolling assumes that the formal array is low-order interleaved, with the first element of the array aligned to a known bank, which then allows the bank number for any element to be computed. Currently, Rawcc always aligns the first element of the array to be on bank 0. If the actual parameter is a pointer to an element that is not known at compile time, the assumption of the code generation phase is violated. Furthermore, even if the alignment of the actual is known, different call-sites may have different actual alignments. When faced with array sections not aligned to a known bank, the compiler has two choices: it can either keep the array distributed and make the formal accesses to it dynamic, or allocate the entire array to one bank retaining disambiguated access. Rawcc currently always does the latter. The former has not been investigated.

A related concept to array sections is partial-dimension passing in C. Note that in C, a multi-dimensional array is really a one-dimensional array of smaller arrays, each with one less dimension. We define partial-dimension passing to be when the actual argument has fewer dimensions than the actual array, and the formal parameter correspondingly has the remaining dimensions. However, unlike array section passing, partial-dimension passing does not require the array to be mapped to a single bank. The reason is that with the padding optimization, the alignment of any partial-dimension argument with bank 0 is maintained.

Chapter 9

Results

This chapter presents some of the results for the Maps compiler system implemented as part of the compiler for Raw. Application programs are compiled through Maps, and simulated on the Raw simulator. Three sets of results are presented. In the first set of results, applications are compiled with and without bank disambiguation, in either case using compiler-exploited ILP. Results show that in the 32-tile case, using bank disambiguation improves performance by a factor of 3 to 5 over not using it for a broad range of programs. In the second set of results, more detailed numbers are collected for programs compiled with bank disambiguation. Speedups are measured for a varying number of tiles; the numbers show that performance scales well with the degree of memory parallelism available. In addition, statistics measuring load balance and aggregate memory bandwidth are presented. In the third and final set of results, the performance benefits of the selective use of dynamic accesses are studied. Results show that software serial moderately improves performance in some cases. Additional results show that independent epochs can be valuable, but only for a high number of tiles, and for a restricted class of programs that have many independent memory references.

Before describing the results, the evaluation framework and application suite are described.

Evaluation framework

Results in this chapter have been obtained using Maps implemented as part of Rawcc, the Raw compiler based on the SUIF compiler infrastructure [24]. Evaluation is performed on a cycle-accurate simulator of the Raw microprocessor, *bug*. The *bug* simulator models an intermediate design point for Raw, roughly modelling the design as of early 1999. The final design for the Raw chip prototype is still in evolution. The design simulated for the results consists of one static network and one dynamic network. Each tile uses a MIPS R4000 instruction set appended with network access instructions for the processing element. The data and instruction memories on each tile are assumed infinite; cache behavior is not simulated. The floating point units simulated are not pipelined. The simulator faithfully models both the single static and single dynamic network, including any contention effects. The experimentally-measured number of cycles for static and dynamic memory references on the simulated design are presented in table 9.1. The numbers in table 9.1 assume no contention, and include both the processing costs and the network latencies.

Distance	0	1	2	3	4
Dynamic store	17	20	21	22	23
Static store	1	4	5	6	7
Dynamic load	28	34	36	38	40
Static load	3	6	7	8	9

Table 9.1: Experimentally measured cost of memory operations in processor cycles for simulated Raw design

The design for the Raw prototype has progressed beyond that in the *bug* simulator and is still evolving; a more recent design point can be found in [34]. Design changes since *bug* include having two identical static networks and two identical dynamic networks, instead of one each. Having two networks of a kind instead of one increases the routing resources available on the chip, *i.e.*, two values can be received at a tile per cycle rather than one. Both networks are register-mapped; two values can be read into a chip in one cycle as the two input operands of a 3-operand MIPS R4000 instruction. An additional motivation for a second dynamic network is better performance for deadlock-avoidance strategies. Another difference of the latest Raw design from the simulated design is that the floating point units are now fully pipelined. A final difference is that the prototype, of course, has limited memory per tile; software caching schemes are being considered for the prototype, rather than hardware caches.

Application speedup is derived from comparison with the performance of code generated by the Machsuif MIPS compiler [42] executed on the R4000 processing element of a single Raw tile. To expose instruction level parallelism across basic blocks, Rawcc uses loop unrolling and control localization [25], among other techniques.

Application suite

Table 9.2 gives the characteristics of the benchmarks used for the evaluation. The benchmarks were derived from the following sources: SPEC [45], Rawbench [2], Jade [46], Mediabench [47], UC Berkeley MPEG-1 Tools [48], UTDSP [49] and CHAOS [50]. In addition, SHA was the version written by Matt Frank at MIT, derived from Schneier's Applied Cryptography [51]. Benchmarks include several dense matrix applications, multimedia applications, and applications with irregular memory access patterns. All the benchmarks are ordinary sequential programs written for a unified address space, without any user directives or pragmas of any kind. Note that we do not rely upon hand-coded parallel programs or programs written for any particular high-performance architecture. All speedups were attained with our automated compiler without any user intervention.

Benchmark	Source	Language	Lines of code	Seq. time (cycles)	Primary Array size	Description
DENSE MATRIX						
Btrix	Nasa7 (SPEC92)	FORTRAN	236	287M	15×15× 15×5	Vectorized Block Tri-Diagonal Solver
Cholesky	Nasa7 (SPEC92)	FORTRAN	126	34.3M	16×16×32	Cholesky Decomposition & Substitution
Swim	SPEC95	FORTRAN	486	96.2M	513×33	Shallow Water Model
Tomcatv	SPEC92	FORTRAN	254	78.4M	32×32	Mesh Generation with Thompson's Solver
Vpenta	Nasa7 (SPEC92)	FORTRAN	157	21.0M	32×32	Inverts 3 Pentadiagonals Simultaneously
Mxm	Nasa7 (SPEC92)	FORTRAN	64	2.01M	32×64, 64×8	Matrix Multiplication
Life	Rawbench	C	118	2.44M	32×32	Conway's Game of Life
Jacobi	Rawbench	C	59	2.38M	32×32	Jacobi Relaxation
Alvinn	SPEC92	C	331	23.8M	30×131	Neural-Network Training
Ocean	Splash/ Jade	C	1174	309.7M	256×256	Ocean Movement Simulation
MULTIMEDIA						
Adpcm	Media-bench	C	295	2.8M	10240	Speech Compression
SHA	Perl Oasis	C	608	1.0M	512×16	Secure Hash Algorithm
MPEG-kernel	UC Berkeley	C	86	14.6K	32×32	MPEG-1 Video Software Encoder Kernel
Latnrm	UTDSP	C	81	103K	64	Normalized Lattice Filter
FIR-filter	UTDSP	C	44	548K	1024	Finite Impulse Response Filter
IRREGULAR						
fppp-kernel	SPEC92	FORTRAN	735	8.98K	-	Electron Interval Derivatives
Moldyn	CHAOS	C	805	63M	256×3, 32000×2	Molecular Dynamics Encoder Kernel
Unstructured	CHAOS	C	850	150M	17377×3, 32000×2	Computational Fluid Dynamics

Table 9.2: Benchmark characteristics. Sequential time is the runtime for uniprocessor code generated by the Machsuif MIPS compiler.

Some benchmarks are full applications; others are key kernels from full applications. Btrix, Cholesky, Mxm and Vpenta are extracted from Nasa7 of SPEC92fp. MPEG-kernel is the portion of MPEG-1 Video Encoder standard [52] that takes up 70% of the total run-time. The code was the UC Berkeley implementation of the standard [48], with the kernel extracted by Sam Larsen at MIT. FIR filter is an important multimedia kernel that is at the heart of several audio, image and video applications. Because the Raw simulator currently does not support double-precision floating point, all floating point operations are converted to single precision.

Some of the applications are modified to compile through the Rawcc compiler. A summary of the modifications follows. Since Rawcc does not yet have automated library support for I/O, the I/O calls in the applications are replaced by simple, low-level I/O primitives supported by the simulator. Some of the applications in which control-flow is not dependent upon data values, data initialized using input is initialized using random data instead. I/O support is currently in the process of being automated in the Rawcc compiler. For many of the applications, the problem sizes are reduced from the standard sizes for the benchmark, in order to reduce simulation time; see table 9.2 for the problem sizes used in the evaluation. Some of the benchmarks had data initialization associated with data declarations – data initialization is not currently automated in Maps; hence the initializations are replaced in the source-code with explicit data assignments.

The rest of this chapter is organized as follows. Section 9.1 compares results with and without bank disambiguation. More detailed results with bank disambiguation are also presented. Section 9.2 presents statistics on memory distribution and utilization for different applications compiled with bank disambiguation. Section 9.3 presents results comparing methods for handling non-disambiguated accesses with methods that disambiguate all accesses. Section 9.4 summarizes the chapter.

9.1 Bank disambiguation results

This section measures the performance of our applications with and without using bank disambiguation. In either case, full compiler-exploited ILP is used. Next, more detailed application results are presented for applications compiled with bank disambiguation, for a varying number of memory banks. The performance of many individual applications are explained.

In the first set of results, figure 9.1 measures the benefits of using the bank disambiguation schemes of this thesis on overall speedups. It compares the speedups on 32 tiles for three strategies: using instruction-level parallelism (ILP) compiler techniques, ILP supplemented with equivalence-class unification (ECU), and ILP supplemented with both ECU and modulo unrolling. The baseline for all three strategies is the sequential program running on one tile with a speedup of one. The compiler techniques for extracting ILP are described in [25], and are conceptually similar to those used for ILP-exposed conventional machines such as VLIWs. The ILP alone numbers, while using 32 PEs, use only one of the memory banks on Raw, as they have no way of distributing memory.

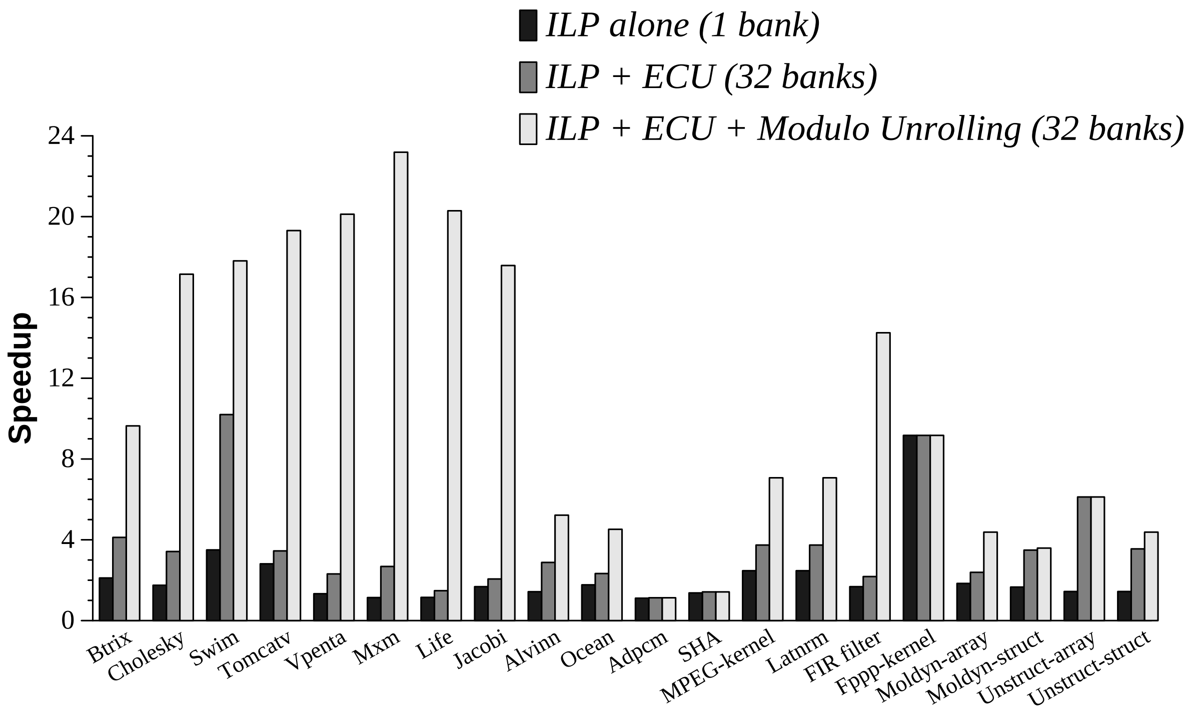


Figure 9.1: Comparison of 32-tile speedups using instruction-level parallelism (ILP) alone, ILP with ECU, and ILP with both ECU and modulo unrolling.

They still use multiple tiles to exploit ILP, and the static network for communication. The latter two numbers use as many memory banks as PEs, and our compiler methods to disambiguate to those banks.

The results show that while using ILP alone often gives a speedup in the range 1-4, using memory parallelism can increase performance substantially beyond that. Specifically, using ECU increases the speedup on average by a factor of two beyond using ILP alone, boosting it to between 2 and 6[1]. In applications where modulo unrolling is applicable, namely dense matrix and some multi-media codes, its use further improves speedup to between 7 and 24. Overall, the methods in this thesis can often deliver an additional factor of 3 to 5 in performance over using ILP alone.

The results in figure 9.1 may have implications beyond Raw. They show that applications with a lot of ILP often have high memory bandwidth requirements. These applications would perform poorly on a system with many functional units but limited memory bandwidth. A Raw machine using ILP alone, and hence using only one memory bank, fits this architectural description, as do superscalars and centralized VLIWs with centralized memory systems. In addition, the Raw machine using ILP alone suffers from high memory latency due to a lack of locality between the processors and the single memory; this latency is analogous to the multi-cycle on-chip wire delays conventional designs will likely suffer in future VLSI technologies. Faced with similar problems, conventional architectures may well find that a software-exposed distributed memory system combined with a Maps-like compiler can improve its performance the same way it improves the performance of a Raw machine with multiple memory banks.

Figure 9.2 breaks down the use of the two techniques for bank disambiguation on arrays. The figure shows for each application, the percentage of arrays whose references are mapped to a single bank, as recommended by ECU, versus those that are distributed, as recommended by modulo unrolling. Arrays can map to a single node if they contain non-affine references; or are reshaped or passed in unaligned sections (see section 8.3).

Detailed application results

Table 9.3 shows the speedups attained by the benchmarks for Raw microprocessors for a varying number of tiles. The numbers are those obtained using both ECU and modulo unrolling. The numbers in the last column, for $N = 32$, are identical to the ILP + ECU + modulo unrolling numbers in figure 9.1. Some application-specific comments are made below.

Dense-matrix applications Most dense-matrix codes got very good speedups, ranging from 5 to 23 for 32 tiles. The predominance of affine function array accesses in such codes implied that all of them benefited greatly from modulo unrolling, as evidenced by figure 9.1. Out of eleven programs, seven attained speedups exceeding 15. The remaining four, Mgrid, Alvinn, Btrix and Ocean, lost some performance for varying reasons. Mgrid

[1] Adpcm, SHA and fppp-kernel are exceptions; see section 9.1 for details.

Benchmark		N=1	N=2	N=4	N=8	N=16	N=32
DENSE MATRIX							
Btrix		0.83	1.48	2.61	4.40	8.58	9.64
Cholesky		0.88	1.75	3.33	6.24	10.22	17.15
Swim		0.88	1.43	2.70	4.47	8.97	17.81
Tomcatv		0.92	1.64	2.76	5.52	9.91	19.31
Vpenta		0.78	1.90	3.36	7.06	12.17	20.12
Mxm		0.94	1.97	3.60	6.64	12.20	23.19
Life		0.96	1.73	3.03	6.06	11.70	20.29
Jacobi		1.01	1.68	3.03	5.95	11.13	17.58
Alvinn		1.04	1.30	2.07	2.93	4.31	5.22
Ocean		0.88	1.16	1.97	3.05	4.09	4.51
MULTIMEDIA							
Adpcm		0.97	0.99	1.19	1.23	1.13	1.13
SHA		0.96	1.18	1.63	1.53	1.44	1.42
MPEG-kernel		0.90	1.36	2.15	3.46	4.48	7.07
Latnrm		0.93	1.30	1.87	2.80	3.39	6.06
FIR-filter		0.80	1.04	1.59	2.55	6.55	14.25
IRREGULAR							
fppp-kernel		0.52	0.73	1.51	3.26	6.72	10.20
Moldyn	array	0.95	1.36	2.38	2.99	4.28	4.38
	structure	0.92	0.94	1.60	2.57	3.11	3.59
Unstruct	array	0.82	1.21	2.35	3.59	5.22	6.12
	structure	0.86	1.29	2.07	3.00	4.10	4.92

Table 9.3: Benchmark speedup with full Maps bank disambiguation through equivalence class unification and modulo unrolling. Speedup compares the run-time of the Rawcc-compiled code versus the run-time of the code generated by the Machsuif MIPS compiler for a varying number of tiles N.

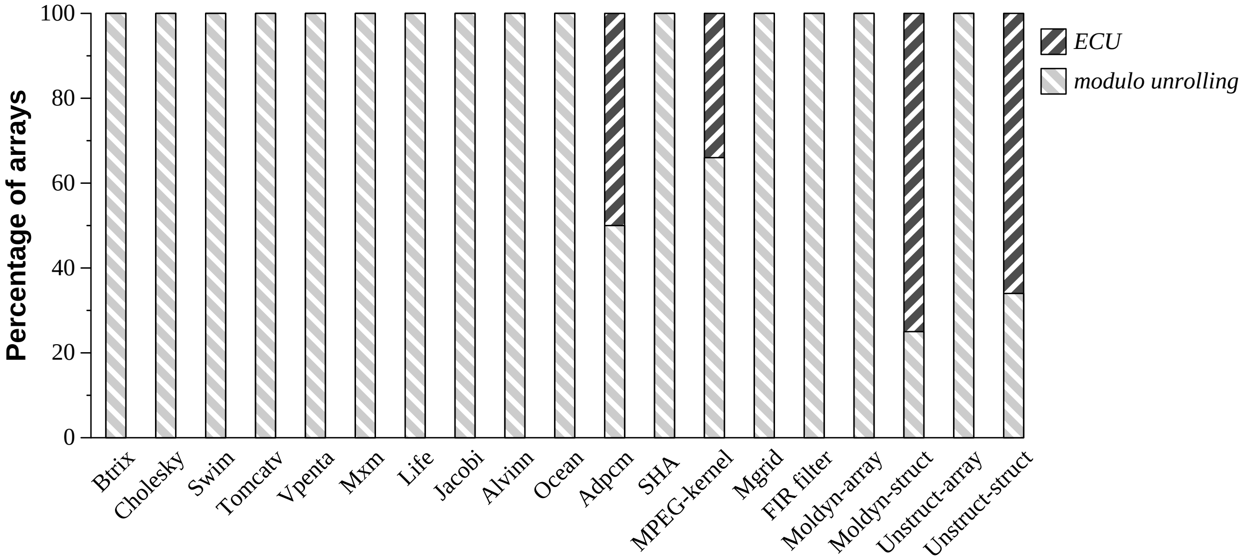


Figure 9.2: Percentage of arrays whose references are disambiguated through modulo unrolling versus those that are disambiguated through equivalence class unification (ECU). fppp-kernel has no arrays, so is not included.

had many of its arrays reshaped, causing them to be allocated on a single node. Alvinn had array accesses in most innermost loops whose indices did not vary with the induction variable of the innermost loop. This caused their memory accesses to all go the same bank. For Btrix, most arrays had a last dimension size of five, which acted as a bound on parallelism available within arrays distributed using the padding optimization. In Ocean, using modulo unrolling caused an N^2 unroll for some loops, leading to excessive code size. To prevent this, the outer loops were not unrolled, forcing some accesses to remain non-disambiguated and slow. Nevertheless, for these four applications, the regular structure of their code ensured competent speedups ranging from 4 to 9, and significant improvements over ILP alone.

Multi-media applications In this class, two applications, Adpcm and SHA, gave low speedups while two, Mpeg-kernel and FIR-filter, gave high speedups. Both Adpcm and SHA have a low degree of available memory parallelism. Furthermore, they were able to exploit only a low degree of ILP, as evidenced by their low speedups in the 'ILP alone' case in figure 9.1. The reason for Adpcm was that the code is inherently serial for the large part: most instructions depend upon the result of immediately previous instructions. For SHA, while some ILP was available, it was too fine-grained for our current techniques to exploit. The communication incurred, even on the fast static network, nullified any parallelism.

Mpeg-kernel and FIR-filter were able to exploit a high degree of speedup. Both made significant use of arrays, and were somewhat similar to dense-matrix codes in program structure. In Mpeg-kernel, a 16x16 window of pixels is manipulated at a time. The speedup was limited to 7 because one of the arrays has a base that is an unknown pointer into a larger array, effectively making its accesses non-affine. The good results on this class of multi-media applications are especially encouraging, as these are key components of the emerging workloads of the future involving audio, image and video data.

Irregular applications Results are presented for Fppp-kernel, Moldyn and Unstructured. Fppp-kernel consists of a time-intensive huge basic-block with mostly accesses to scalar data. A high degree of ILP is exploited. Since only scalar variables are present, the space-time scheduler itself is able to distribute data in the ILP-alone number; there is no further improvement from ECU and modulo unrolling. Moldyn and Unstructured are more typical examples of scientific programs with irregular access patterns. Their accesses are predominantly non-affine. Available parallelism is within different fields of array elements. For both, we present results for two different versions: using structures and using arrays. While the structure versions use arrays of structures to represent their data, the array versions use 2-dimensional arrays with the second dimension representing the fields of the structure. The opportunities for memory parallelism are identical for both versions, but Maps exposes that parallelism through different means. Memory parallelism in the array versions is exposed through array distribution and modulo unrolling, while parallelism in the structure version is exposed through equivalence class unification. The different speedups for the two versions are accounted for by details concerning address calculation costs and opportunities for optimization of those costs. Note that the speedup for Moldyn is obtained without special handling of the reduction in its time-intensive loop. Its performance should improve further with reduction recognition.

Admittedly, at this point our coverage of irregular applications is low. Many irregular applications have small basic-blocks. Even when memory parallelism is available in terms of objects, it may not be exploitable because of a low degree of parallelism within basic blocks. Ongoing research in the Raw group is investigating techniques to exploit ILP across basic blocks and through speculation.

9.2 Memory distribution and utilization

This section measures statistics concerning memory distribution and utilization for our applications. Two sets of numbers are presented, measuring memory load balance and memory bandwidth utilization.

Memory load balance

The first set of numbers measures memory load balance. In general, balanced data distribution is desirable because it minimizes the per-tile memory needed to run an application, and it alleviates the need to build large and centralized memory which is also fast.

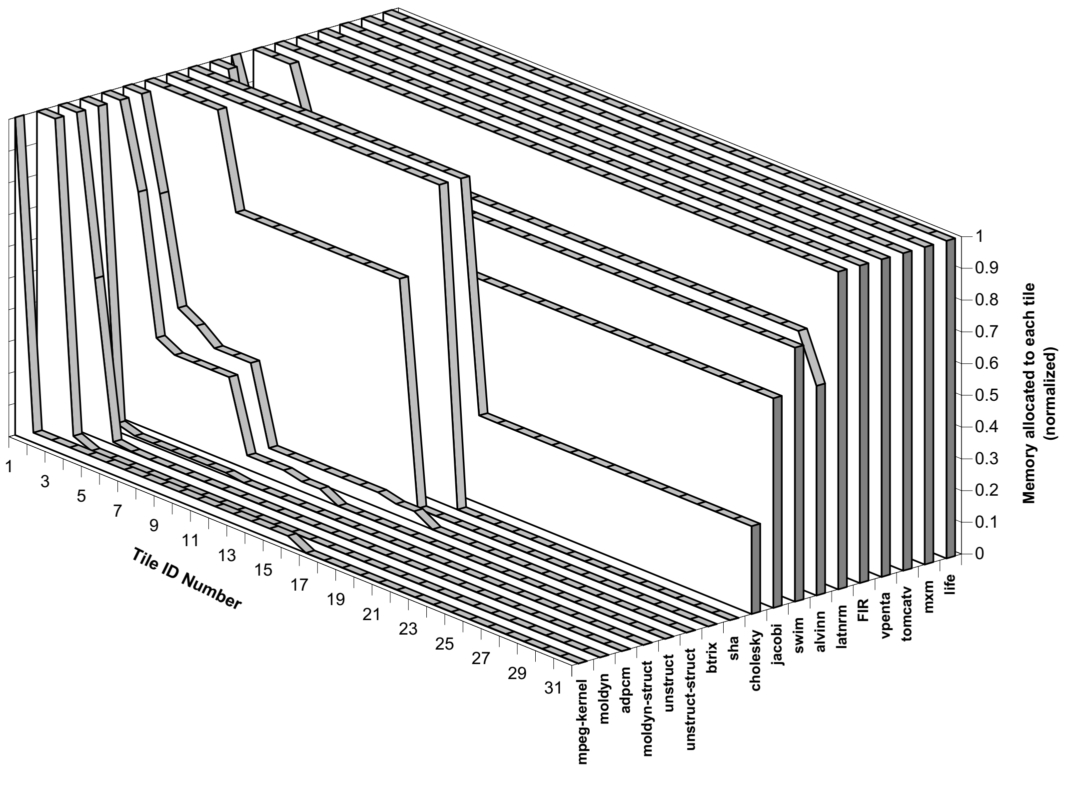


Figure 9.3: Distribution of primary data on a 32-tile Raw machine. The tiles are sorted in decreasing order of memory consumption. For each benchmark, the graph displays the memory consumption on each tile normalized by the memory consumption of the tile with the largest consumption. (**Needs updating with new benchmarks)**

Figure 9.3 shows the distribution of primary data across tiles for our benchmarks executing on 32 tiles. Most dense matrix codes can fully distribute their data; Swim and Cholesky can only partially distribute their data because of their small problem sizes, but their distributions become balanced with larger problem sizes. For smaller data sets, the array dimension sizes are comparable to, or smaller than, the number of tiles (32 in figure 9.3); since modulo-unrolling low-order interleaves the values on the last dimension value, a small dimension size results in a slight load imbalance. MPEG-kernel has a few non-affine array accesses; therefore, it cannot use only disambiguated memory accesses while distributing data. Consequently, it handles one array using ECU; hence its distribution is not load-balanced. The remaining applications can partially distribute their data across a range of 3 to 16 tiles. For many of these remaining applications, the limited load balance is because of two factors: the limited number of equivalence classes, and the unequal size of the classes.

Memory bandwidth utilization

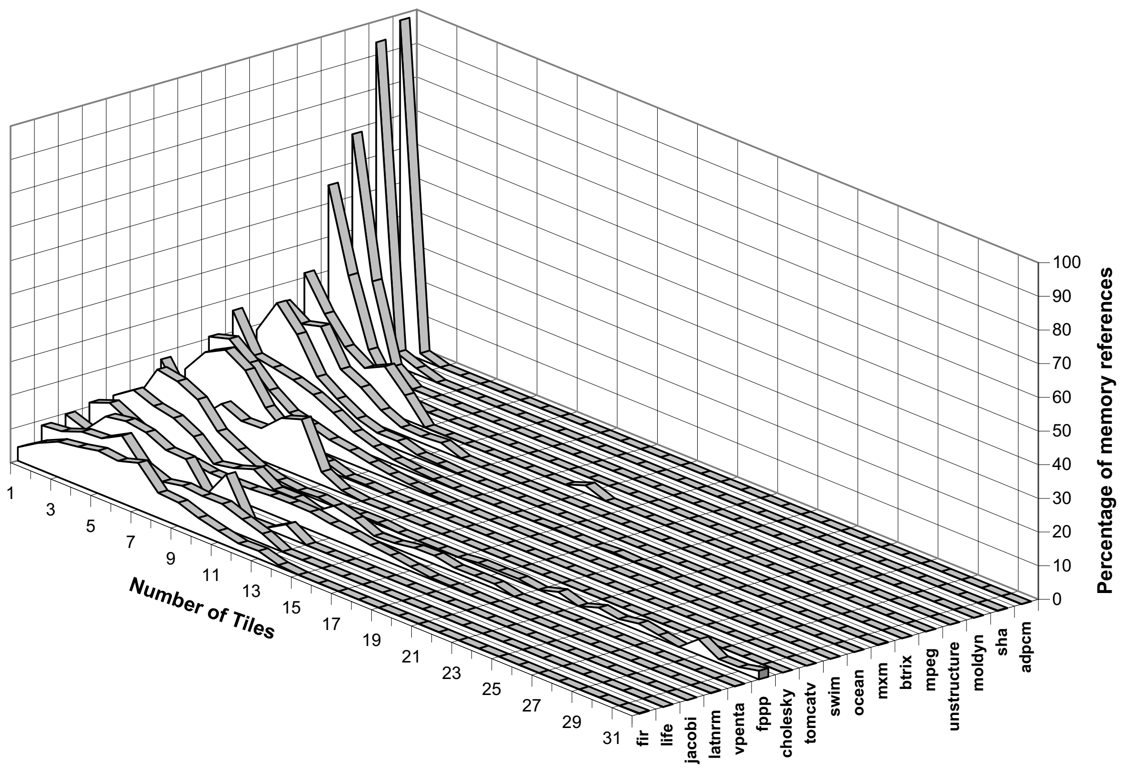


Figure 9.4: Weighted bandwidth utilization of the memory system on a 32-tile machine. The graph displays the percentage of memory references being issued in a time slot when a given number of tiles is issuing memory requests.

Here, memory bandwidth utilization is measured when bank disambiguation is used.

Memory bandwidth utilization measures how well an application takes advantage of Raw's independent memory banks. It depends on the amount of memory parallelism exposed by Maps and the amount of parallelism in the application.

Figure 9.4 measures the weighted memory bandwidth utilization of a 32-tile machine. It plots the percentage of memory references being issued in a clock cycle when a given number of tiles is simultaneously issuing memory requests. Therefore, the sum of the percentages for any one application is 100%. For example, figure 9.4 shows that for Cholesky, almost 20% of the memory references are issued in a cycle in which a total of 7 memory references issued in all the tiles. Results show that except for the two highly serial benchmarks (Adpcm and SHA), all the benchmarks are able to exploit at least a small amount of parallel memory bandwidth. Most of the other multimedia applications and dense matrix applications have at least 20% of their accesses being performed on cycles that issue five or more accesses, with Vpenta enjoying 10-way memory parallelism for over 20% of its accesses.

The significance of figure 9.4 is that for most applications, it shows that it is not true that 1 or 2 memory instructions are all that are needed to exploit available memory parallelism. The figure shows that a higher degree of memory parallelism is often available in applications; further, that the available memory parallelism can be exploited on architectures that support multiple memory instructions per cycle.

9.3 Static vs. dynamic accesses

This section investigates whether the selective use of dynamic access can improve performance. First, the performance of 3 of our applications is compared for three cases: with only static accesses, with software serial ordering, and with independent epochs and updates. Second, a case when software serial ordering improves performance is demonstrated.

Static (disambiguated) accesses are usually better than dynamic (non-disambiguated) accesses because of their low overhead. Sometimes, however, static accesses can only be attained at the expense of memory parallelism. MPEG-kernel, Unstructured, and Moldyn are benchmarks with irregular accesses that can take advantage of high memory parallelism. This section examines the opportunity of increasing the memory parallelism of these programs by distributing their arrays and using dynamic accesses to implement parallel, irregular accesses.

Figure 9.5 shows the performance of the aforementioned benchmarks when all arrays are distributed. Irregular accesses are implemented through dynamic accesses, with software serial ordering to ensure correctness. Results for Moldyn and Unstructured are poor, with slow-downs for all configurations. MPEG-kernel attains speedup but is twice as slow as its purely static speedup. This result is not surprising: dynamic accesses serialized through a turnstile are slower than corresponding static accesses serialized through a memory node. In either case, the degree of serialization is the same, with the dynamic case suffering from additional overhead.

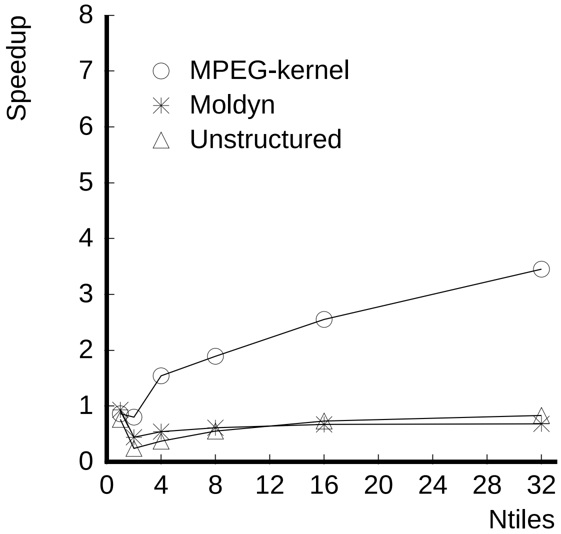


Figure 9.5: Benchmark speedup with all arrays distributed, with irregular array references implemented as dynamic accesses with software serial ordering.

To improve the performance with dynamic accesses, serialization of dynamic accesses has to be reduced through independent epoch and update optimizations. Currently, independent epoch generation has not been automated, so our evaluation uses a hand-coded implementation of independent epochs and updates. To simplify the hand-coding task, the optimizations are limited to a selected loop from each of Moldyn and Unstructured, and the full MPEG-kernel. The loop selected from Moldyn accounts for 86% of the run-time. In Unstructured, many of the loops with irregular accesses have similar structure; one such representative loop is selected. Figure 9.6 shows the performance of dynamic references when independent epoch and update optimizations are applied to these applications, compared with the unoptimized dynamic performance and the static performance. The figure shows that the dynamic optimizations are effective in reducing serialization and attaining speedup. All three benchmarks benefit from independent epochs, while Moldyn and Unstructured benefit from updates as well. Together, the optimizations completely eliminate the turnstile serialization for these applications.

The speedup trends of these applications reflect the amount of available memory parallelism. For static accesses, the amount of memory parallelism that can be exposed through ECU is limited to the number of alias equivalence classes. Depending on the access patterns, the amount of useful memory parallelism may be less than that. This level of memory parallelism does not scale with the number of tiles. For a small number of tiles, ECU can expose enough parallelism to satisfy the number of processing elements. For a larger number of tiles, insufficient memory parallelism causes the speedup curve to level off.

In contrast, the use of dynamic accesses allow arrays to be distributed, which in turn

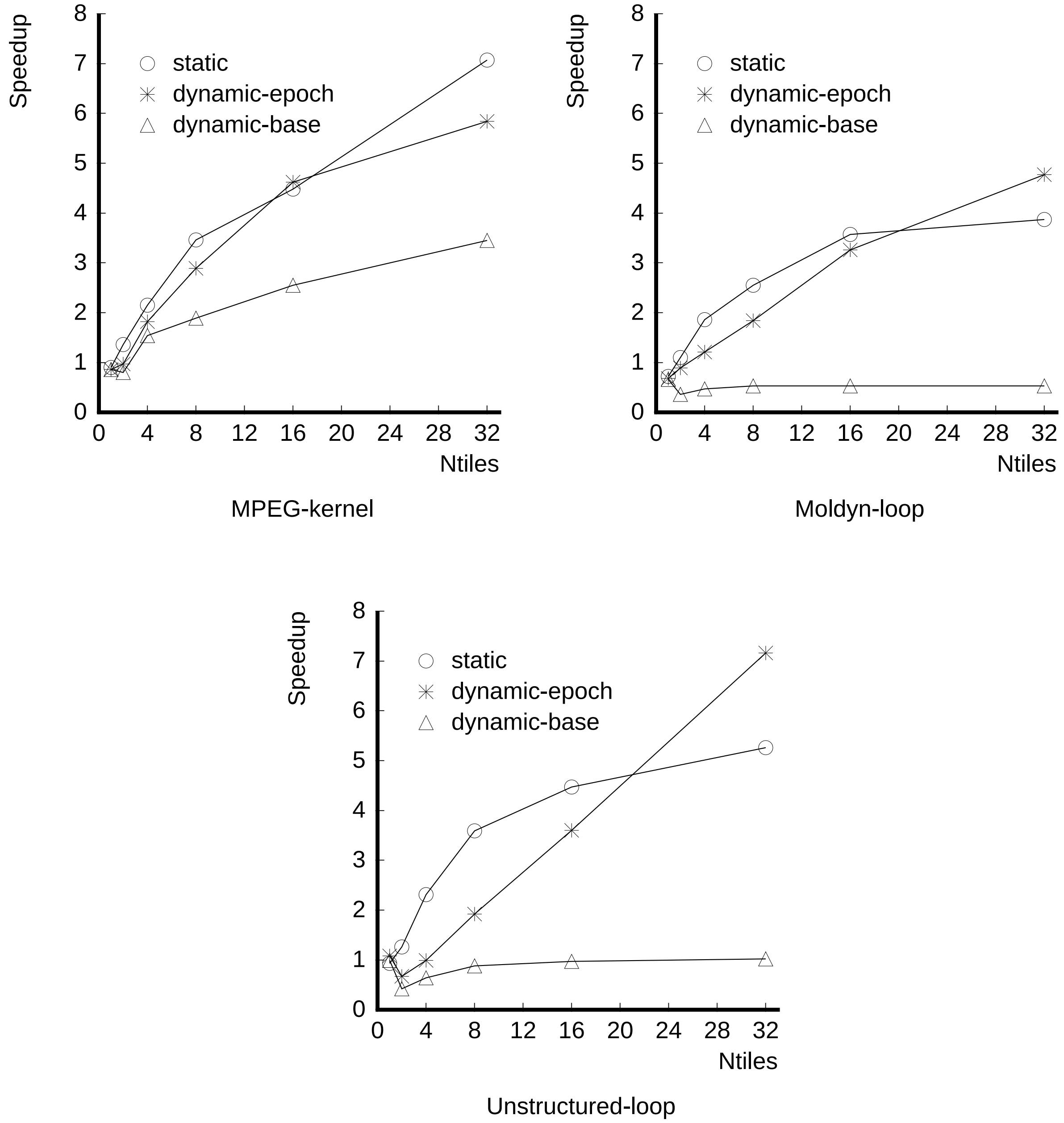


Figure 9.6: Speedups of benchmarks with selective use of dynamic accesses. The *dynamic-epoch* numbers are for the programs optimized with independent epochs and updates. The dynamic case without optimization is in *dynamic-base*. For comparison the static number, with no dynamic accesses, is also presented.

exposes memory parallelism scalable with the number of tiles. As a result, the speedup curve for optimized dynamic scales better than that for static. For up to 16 tiles, static outperforms optimized dynamic; for 32 tiles, optimized dynamic actually outperforms static, and the trend suggests that optimized dynamic increasingly outperforms static for even larger numbers of tiles. To understand this trend, note that the static case has lower memory parallelism but also lower overhead for each access; for a lower number of tiles, the available parallelism is adequate. Dynamic accesses perform poorly for a low number of tiles because the available parallelism is not fully exploited by the low number of tiles, yet the overhead of dynamic accesses causes them to be slower. For a high number of tiles, the increased parallelism of dynamic accesses enables them to partially offset the performance loss from their higher overhead. For the dynamic experiment, only the irregular accesses are selectively made dynamic; the affine array accesses and all scalar data are accessed on the static network.

Why do we need software serial ordering?

As discussed earlier in this section, dynamic accesses using software serial ordering can never perform better than static accesses disambiguated through ECU. Here we show, by using an example from Unstructured, how software serial ordering can nevertheless be useful.

Unstructured contains an array $X[]$ that is accessed in only two loops, an initialization loop (*init*) and a usage loop (*use*). The initialization loop makes irregular accesses to $X[]$ and is executed only once. The usage loop makes affine accesses to $X[]$ and is executed many times. For best performance, Maps should optimize the placement of $X[]$ for the usage loop.

Array mapping	Loop	Access type	Speedup
centralized	init	static serial	1.89
	use	static serial	3.86
	total	–	3.85
distributed	init	dynamic serial	0.59
	use	static parallel	4.43
	total	–	4.42

Table 9.4: An example of overall performance improvement through the use of software serial ordering. Software serial ordering enables Maps to distribute a critical array, which optimizes for static parallel access in the critical *use* loop in exchange for dynamic accesses with software serial ordering in the non-critical *init* loop. Performance is measured for 32 tiles.

Table 9.4 compares the performance of the loops when $X[]$ is placed on one tile to when it is distributed across 32 tiles. When the array is centralized, both *init* and *use* attain speedups because they enjoy fast static accesses. When the array is distributed, however, *init* suffers slowdown because it has dynamic serial accesses going through a turnstile, while *use* attain better speedup compared to the centralized case. For the full program, however, the performance of *use* matters much more. Thus, distributing $X[]$

provides the better overall performance, despite the overhead *init* incurs from software serial ordering.

The example in table 9.4 illustrates the general use of software serial ordering (SSO). SSO is a way of enforcing dynamic dependences that is more efficient than other mechanisms such as complete serialization or placing barriers between the dependent accesses. SSO is used not to improve the performance of the code segment using it, but as an enabling mechanism to allow the compiler to improve the parts of the program that really affect performance. SSO provides a universal and efficient handling of dynamic accesses in the absence of applicable optimizations. The overall utility of dynamic accesses remains to be seen, but its use with software serial ordering provides a reasonable starting point on which further optimizations can be explored.

9.4 Summary

This chapter describes experimental results on executables produced from sequential code using the Raw compiler, and evaluated on the Raw simulator. The suite of applications spans dense-matrix applications, multimedia applications, and irregular applications. To measure the benefit from bank disambiguation, the speedup of each application is measured on a 32-tile machine using ILP alone, and ILP augmented with bank disambiguation. A speedup of 1 corresponds to the application running on a machine with one processing element and one memory bank. The results show that while ILP alone provides some speedup, augmenting ILP with bank disambiguation improves the speedups by an additional factor of 3 to 5 for many applications over using ILP alone. The speedups from ECU and modulo unrolling are separated out to measure their individual impacts. More detailed results show how the speedups using ILP augmented with bank disambiguation scale with power-of-two tile numbers between 1 and 32.

The next part of the chapter measures more parameters for the bank disambiguated codes. First, results are presented on the load balance provided by the different applications. Maps is able to provide excellent load balance when modulo unrolling is used, and some load balance when ECU is used. Using ECU, the load balance is limited by a limited number of equivalence classes and unequal sizes of the different classes. Second, results are also presented showing the frequency of cycles in which multiple memory requests are issued in one cycle. Results show that modulo unrolling delivers a high degree of memory parallelism; ECU delivers some memory parallelism, but a lesser amount.

The final part of the chapter measures the benefit from the selective use of dynamic accesses, over using purely static schemes. Overall, the results show that while in some cases performance improves, the improvement is not large. While more extensive evaluation might show otherwise, our results show that using dynamic accesses outperforms purely static schemes in some cases only when independent epochs can be used, and only for 32 or more tiles. As expected, SSO leads to poor performance in the parts of the code it is used. Results on one of our applications shows, however, that SSO allows the rest of the program to run faster, improving overall performance.

Chapter 10

Related work

This chapter discusses related work with respect to the compiler methods in this thesis. Compiler methods having similarities with, or relevance to, bank-exposed compilers are discussed. Compilers investigated include compilers for other architectures, such as microprocessors, DSP chips, multiprocessors, vector machines and systolic arrays. Features of architectures are discussed only to explain the role of their compilers; for direct comparisons of software-exposed architectures to other architectures, see [1].

This chapter is organized as follows. Section 10.1 discusses bank disambiguation strategies proposed elsewhere. Section 10.2 describes four kinds of memory disambiguation, other than bank disambiguation, developed in the literature. Section 10.3 describes a hardware-supported addressing scheme for DSP chips called modulo addressing, and its relation to compilers for software-exposed machines. Section 10.4 discusses multiprocessor compilers. Section 10.5 discusses compilers for vector machines. Section 10.6 discusses compilers for systolic arrays.

10.1 Bank disambiguation

This section describes work related to bank disambiguation in both general-purpose and DSP domains. Little work has been done on bank disambiguation in the general-purpose community, as few bank-exposed architectures have been proposed in the past. Indeed, that bank disambiguation is considered a hard problem is probably why few general-purpose bank-exposed architectures have been proposed. Recently, however, proposals have appeared for bank-exposed architectures, for example, Cooper and Harvey [53] propose a small, fast, exposed memory for general-purpose architectures to handle memory traffic from register spills. In essence, the scheme in [53] bank disambiguates all register spills to one bank. DSP chips, on the other hand, have long used more than one exposed memory bank. The lack of bank disambiguation methods has meant that most DSP chips have been hand-coded using assembly language. Recently, some work has been done for bank disambiguation for DSP chips, and is described in this section.

An early general-purpose bank-exposed design is the ELI-512 VLIW machine [16] proposed by Josh Fisher in 1983. The ELI-512 is an unusual VLIW design with a point-to-point network connecting its various tiles, each with its exposed memory bank. Fisher recognized the importance of bank disambiguation for such a machine. The ELI-512 provided two ways to access memory: through a fast "front door" to directly to a memory bank when the accessed bank is known at compile-time, or a slower "back door" for accesses for which the accessed bank is unknown. These mechanisms identify the architecture as bank-exposed. Most VLIWs today, however, use global buses and a unified memory system rather than point-to-point networks.

Modulo unrolling is related to an observation made by Fisher [16]. He observes that unrolling can sometimes help disambiguate accesses. Based on this observation, his compiler relies on user annotations to determine the unrolling factor needed for such disambiguation. In contrast, modulo unrolling is a fully automated and formalized technique that computes the necessary unrolling factors needed to perform disambiguation for dense matrix codes. It includes a precise specification of the scope of the technique and a theory to predict the minimal required unroll.

Saghir, Chow and Lee [54] have developed a method for exploiting dual-memory-bank digital-signal processing (DSP) chips. Many DSP chips today have two exposed memory banks, called X and Y memory, such as in the Motorola DSP56000 family [18] and the NEC μPD7701x family [55]. The method in [54] divides the data in C programs among the two banks in a way that optimizes for memory parallelism. The method consists of two steps. First, an interference graph is built whose nodes are the variables in the program, and each edge represents the parallelism penalty of putting the two variables connected by the edge on the same bank. The penalty is initialized to zero, and incremented for every pair of references that can issue in parallel. The increment amount is the loop-nesting depth of the references, roughly approximating the importance of placing the references on different banks. In the second step, the interference graph is partitioned among the two banks using a greedy algorithm that aims to minimize the total cost of the edges within each bank. In this way, the parallelism penalty incurred is minimized.

The method in [54], described in the previous paragraph, is bank-disambiguation, in that it assigns every variable to a bank. It is superficially similar to ECU, however, it is complementary to our work. The method in [54] is more akin to virtual-to-physical bank assignment, rather than ECU. Virtual-to-physical bank mapping is done by the space-time scheduler [25], the second phase of Rawcc after Maps, in a manner that places data on different banks if there are many parallel pairs of accesses to the data. The differences of Maps with [54] are as follows. First, ECU derives the virtual-bank partition as the underlying upper-bound on the parallelism available, while [54] targets a particular number of physical banks, taking into account the actual parallel references present[1]. Second, ECU handles programs with arbitrary pointers, while [54] does not. ECU, in fact, uses pointer analysis to guide the placement of data. Third, [54] always

[1] One shortcoming of ECU compared to [54] is that ECU says nothing about how mapping to physical banks is done. ECU delegated the responsibility for virtual-to-physical mapping to the space-time scheduler. The space-time scheduler, in turn, could use an approach similar to [54].

places each data object on a single bank, including arrays and structures. In contrast, modulo unrolling distributes arrays, and ECU distributes structures, thus extracting parallelism within objects. For many dense-matrix and signal-processing applications, extracting parallelism within arrays using modulo unrolling greatly improves performance (see chapter 9); [54] has no analog for modulo unrolling.

Sudarsanam and Malik [56], in an earlier work, present a solution to the same problem as in [54]. The innovative feature about the method in [56] is that it handles register allocation and memory bank assignment simultaneously, instead of in a de-coupled manner. Most other DSP compilation strategies [54, 57, 58] perform register allocation and memory bank assignment separately. Simultaneous handling of register allocation and memory bank assignment is valuable in architectures where there are constraints on which registers can be used to store data from a particular data bank. Many DSP architectures [18] place such constraints because of their limited connectivity. The method in [56] models the problem using a constraint graph, on which graph labeling is performed to find a register and memory assignment. In terms of a comparison with Maps, since the method in [56] is conceptually similar to the method in [54], the same comparison comments for [54] with Maps apply to the comparison of [56] with Maps. The only additional observation is that Maps performs data and instruction placement (virtual-to-physical mapping) simultaneously, as in [56], in the space-time scheduler. This is because Raw has heterogeneous connectivity: register files are directly connected only to their local memory banks, making it profitable to consider data and instruction placement together.

Systolic arrays have exposed memory banks, however, their computation model is entirely different from our model. Section 10.6. discusses systolic compilers.

10.2 Other kinds of memory disambiguation

Atleast four different kinds of memory disambiguation, other than bank disambiguation, have been proposed in the literature. These are relative memory disambiguation, run-time disambiguation, dynamic memory disambiguation, and affine-memory disambiguation. Each of the four is discussed in this section, in the order listed. The discussion states what they are, how they compare to bank disambiguation, and if they are useful for Maps in any way.

Relative memory disambiguation [59] is a kind of memory disambiguation useful for any machine with multiple memory banks that allows multiple outstanding requests to memory. The work in [59] targeted bus-based VLIW machines such as the Multiflow Trace [31]. Relative memory disambiguation aims to discover cases where two memory accesses never refer to the same memory location. Successful disambiguation implies that the accesses can be executed in parallel. Relative memory disambiguation uses dependence analysis, pointer analysis and dataflow analysis techniques to discover the relative offsets of different references. For example, relative memory disambiguation may be able to prove that the references $*p$ and $*(p+1)$ always go to different banks if the data is low-order interleaved. To do so, the compiler would need to prove that p is not modified in between the two accesses. Dataflow analysis may be able to prove that p is not modified.

Relative memory disambiguation does not specify the actual bank that a reference accesses, or guarantee that the bank accessed is the same in all dynamic instances, hence relative memory disambiguation solves a different problem from bank disambiguation. Nevertheless, relative memory disambiguation is useful in Maps in the independent epoch optimization for non-disambiguated accesses. An Independent epoch may be used for a set of non-disambiguated accesses when all the accesses are independent. Relative memory disambiguation is able to prove independence of accesses in some cases.

Run-time disambiguation, proposed by Nicolau [60], is another kind of memory disambiguation. Nicolau's method uses software address-comparison checks, inserted into the code by the compiler, that compare addresses for different memory references at run-time. If the check reveals that the references access different addresses, then a code fragment that re-orders them is run, else a slower fragment using in-order accesses is run. Thus, run-time disambiguation aims for scheduling flexibility of instructions; re-ordering instructions leads to performance gain, as early scheduling of instructions having many dependent-instructions increases the ILP exposed. Run-time checks are used only when purely static dependence-analysis fail to prove independence, as run-time checks incur overhead, and the number of checks grows rapidly with the number of instructions re-ordered: if m loads bypass n stores, mn run-checks are needed.

We initially considered run-time disambiguation as a scheme to efficiently handle dynamic accesses, before discovering software serial ordering. The proposal involved comparing addresses of references using run-time checks; if different the references would be issued in parallel, else they would be serialized. We abandoned this possibility in favor of software serial ordering (SSO) upon discovering SSO. Software serial ordering, like run-time disambiguation, overlaps much of the latency of dynamic accesses while ensuring dependence. Unlike run-time disambiguation, however, SSO is more scalable as it uses no explicit check of addresses, and does not incur their overhead. The number of run-time checks in run-time disambiguation grows quadratically with the number of accesses involved. Further, SSO, unlike run-time disambiguation, overlaps much of the latency of accesses even when they are dependent.

Run-time disambiguation, discussed above, is a different problem from bank-disambiguation, and complementary to it. Run-time disambiguation need not to predict the bank accessed by each reference, so it does not. Instead, run-time disambiguation is complementary to bank-disambiguation: it is conceivable to use run-time disambiguation between non-disambiguated accesses, or accesses disambiguated to the same bank, to re-order them for scheduling flexibility[2].

Dynamic memory disambiguation, such as in Chen [61] and Gallagher *et. al.* [62], is the hardware version of run-time disambiguation. Like run-time disambiguation, dynamic memory disambiguation performs checks of addresses at run-time to increase scheduling

[2] Run-time disambiguation is not needed between accesses disambiguated to different banks, as they are already known to be independent.

flexibility, and thus ILP. Unlike run-time disambiguation, dynamic memory disambiguation does not perform software-checks; rather, it uses a special hardware unit called a *memory conflict buffer* to perform the checks in hardware. The memory conflict buffer was proposed in [61] and [62], and improved in [63]. To use the memory conflict buffer approach, the software must use a special load instruction, called a *preload*, for any load that has been moved from after possibly conflicting stores, to before. Further, the software must insert a special *check* instruction to signal to the hardware to look for dependence violations, and jump to compiler-provided correction code to recover from the violations.

Since dynamic memory disambiguation is similar to run-time disambiguation in the task performed, in comparing with bank disambiguation, the same comments apply for dynamic memory disambiguation as for run-time disambiguation. That is, dynamic memory disambiguation solves a different problem from bank disambiguation; dynamic memory disambiguation does not ensure that accesses go to a single bank. Instead, it is complementary to the bank disambiguation methods in Maps: dynamic memory disambiguation may be used to re-order non-disambiguated accesses, or accesses disambiguated to the same bank.

Affine-memory disambiguation, a term used by Maydan [64] for his work, is yet another kind of memory disambiguation appearing in the literature. A compile-time analysis, affine-memory disambiguation is a precise form of dependence analysis. Traditional dependence analysis [65, 66, 67] aims to prove that pairs of references are independent, failing which it conservatively assumes that the references may be dependent, even if at run-time they turn out to be always independent. Affine-memory disambiguation aims to answer the question of dependence more precisely; *i.e.*, it places a dependence edge if and only if there is an actual dynamic instance at run-time for which the references access the same address. Maydan [64] claims that, for all cases appearing in practice, his analysis is exact.

Affine-memory disambiguation differs from bank disambiguation. Bank disambiguation predicts the bank number for any reference; affine-memory disambiguation, a form of dependence analysis, proves that pairs of references are independent. Maps, like most compilers, benefits from dependence analysis. Affine-memory disambiguation can be used for dependence analysis in Maps, instead of the more traditional methods used.

10.3 Modulo addressing and streaming applications

Modulo addressing [68, 69] is the name of a hardware-supported addressing mode often present in DSP chips, such as the NEC μPD7701x [55]. Modulo addressing maps the software-address *addr* to always implicitly refer to location $addr\%M$, where M is an integer, usually a power of two. In effect, modulo-addressing implements the address space as a circular buffer. Modulo addressing is useful in applications such as FIR filter [68], where only a sliding window of a large stream of data is kept on chip at one time. Without modulo-addressing, the entire array needs to be shifted by one element every time a new element comes in from the stream. With modulo addressing, no shift is required, as the new element simply overwrites the oldest (outgoing) element at the end of the circular buffer.

Modulo addressing is mentioned here for two reasons. First, to clarify that modulo addressing has nothing to do with modulo unrolling, with which it bears a similar name. Second, there is an interesting possibility of using software-implemented modulo addressing in Raw for real-time streaming applications. The FIR-filter evaluated in the results chapter (chapter 9) implements a version where the entire data-stream is mapped to unique address-space locations at compile-time. For a real-time, potentially infinite, streaming version of FIR, mapping the entire data stream to unique locations is not possible. Ordinarily, a real-time FIR on Raw requires explicit data-shifts to accommodate new elements, leading to a whole new, streaming compilation model incompatible with Rawcc. With modulo addressing, however, no shift is required, as the new element is mapped to an existing location at the middle of the array, where the outgoing element used to reside. Moreover, with small modifications, this modulo addressing compilation model is *compatible* with Rawcc. Before access, references to the data stream would perform a mod operation of the array index with the circular buffer size, thus mapping an infinite stream to a finite buffer. The circular buffer size need not be equal to N, the number of banks; instead it can be any power-of-two: mod operations with powers of two are cheaply implemented in software as bit-wise *and* operations.

10.4 Multiprocessor compilers

Other researchers have parallelized some of the benchmarks in this paper for multiprocessors. Automatic parallelization has been demonstrated to work well for dense matrix scientific codes [22, 23]. In addition, some irregular scientific applications can be parallelized using the inspector-executor method [70]. Typically these techniques involve user-inserted calls to a runtime library such as CHAOS [71], and are not automatic. The programmer is responsible for recognizing cases amenable to such parallelization, namely those where the same communication pattern is repeated for the entire duration of the loop, and inserting several library calls.

In contrast, the Maps approach is more general and requires no user intervention. Its generality stems from its exploitation of ILP rather than coarse-grain parallelism targeted by [70] and [22]. Multiprocessors are mostly restricted to such coarse-grain parallelism because of their relatively high communication and synchronization costs. Unfortunately, finding coarse grain parallelism often requires whole program analysis by the compiler, which works well only in restricted domains. A software-exposed machine can successfully exploit ILP because of the presence of fast register-level communication. For the Raw machine, this is the static network. Of course, software-exposed machines can exploit coarse-grain parallelism as well if the compiler so chooses.

In addition to the advantages of ILP, cheaper synchronization is another benefit of a fast compiler-scheduled network in software-exposed architectures such as Raw. In multiprocessors, sharing a variable across different tiles requires heavy-duty synchronization mechanisms such as locks to ensure correct serial ordering. When a compiler-scheduled network is present, however, the ordering guarantees provided by the network provide correctness, without the need for mechanisms such as locks.

For message-passing multiprocessors, it is profitable to know the bank numbers accessed for data elements at compile-time. Compile-time knowledge avoids the need for a software check at run-time of whether the element is remote or local. Remote elements are accessed via user-level messages[3]. Much work has been done in the area of data mapping and code generation for multiprocessors [72, 73, 74, 75, 76, 77]. These methods decompose the program into several parallel threads, one per processor, and decompose arrays using custom allocation strategies, such as blocked, cyclic, or block-cyclic, depending upon which allocation strategy minimizes communication for that program. Further, methods such as [73] specify, for each processor, the accessed data element regions on each remote processor's memory. [73] models the accessed data regions on remote processors by elements satisfying a system of linear inequalities in the data space.

While methods for affine-function code generation on multiprocessors generate bank information, they are inappropriate for use on a bank-exposed microprocessor for the following two reasons. First, multiprocessor compilation methods often produce block or block-cyclic data partitions, which provide coarse-grained parallelism, not ILP. In many programs array accesses that are spatially close, such as $A[i]$ and $A[i+1]$, are also temporally close. If $A[]$ is block-partitioned, sets of temporally close loop iterations would likely be mapped to the same tile, rather than different tiles, thus providing no ILP. Since bank-exposed machines rely on ILP, block or block-cyclic partitions are inappropriate for such machines. Second, forcing multiprocessor compilers to only use cyclic partitions (low-order interleaving) is not a good strategy for generating code for bank-exposed architectures. Bank-exposed architectures need bank disambiguation for good performance; a reference is bank disambiguated if it accesses the same bank in every dynamic instance. Multiprocessor compilation schemes generate remote-region information for each remote bank accessed, *i.e.*, for each tile, the region of data accessed on every remote bank by the current tile is output as symbolically defined region in the data space. Multiprocessor compilers do not generate bank-disambiguated code; however, it is conceivable to adapt them to do so. One possibility is grouping remote-region accesses together to run in the program code, instead of bulk-message handlers as in multiprocessors. Each access in the remote region accesses a single tile (its home), thus providing bank disambiguation. While conceivable, grouping the remote regions together is undesirable for the following two reasons. First, remote-region sizes are unknown at compile-time when unknown loop bounds are present. Unknown communication patterns make it difficult to use register-level communication, such as the compiler-routed

[3] Shared-memory multiprocessors do not need to do a software local vs. remote check, as shared-memory hardware does the check for every element anyway. Nevertheless, shared memory multiprocessors too benefit from knowledge of the location of data, in order to map computation in a manner than minimizes communication.

communication on Raw. Second, multiprocessor compiler methods provide no way of integrating non-affine accesses into their parallelization strategies, while modulo unrolling integrates seamlessly with non-affine parallelism (from ECU) and with ILP.

Software distributed shared memory schemes on multiprocessors (DSMs) [78] [79] are similar in spirit to Map's software approach to managing memory. They emulate in software the task of cache coherence, one that is traditionally performed by complex hardware. In contrast, Maps turns sequential accesses from a single memory image into decentralized accesses across Raw tiles. This technique enables the parallelization of sequential programs on a distributed machine. The question of cache-coherence does not arise in bank-exposed architectures as they do not use global caches, *i.e.*, each memory location can be cached in only one fixed cache bank, on which its address is mapped.

10.5 Compilers for vector machines

Vector machines aim to exploit fine-grained memory parallelism in vectorizable scientific codes [32, 80]. Vector machines provide *vector instructions*: vector instructions perform the same operation in successive cycles on successive elements of *vector registers*. Only one operation is executed per cycle from each vector instruction. No parallelism is exploited within vectors [81]; rather, parallelism stems from two factors: first, parallelism between different vector instructions and, second, parallelism between vector and scalar instructions. Hardware-supported methods like chaining and tailgating allow a vector instruction to start before all the operations of a preceding vector instruction, on which the current instruction is dependent, complete. Even with chaining and tailgating, however, only one operation is executed per cycle from each vector instruction.

Memory is accessed in vector machines using vector-memory instructions; vector-memory instructions access successive locations that form an arithmetic progression, *i.e.*, are related by a constant stride. Vector machines typically use many interleaved main-memory banks and no caches. While only a single word request is issued per cycle, the long access latencies of successive words are partially overlapped provided they go to different banks. If addresses are hardware-mapped in a low-order interleaved manner across the banks, a stride of one ensures that successive accesses go to different banks. For strides that are powers of two, however, low-order interleaving results in the same bank being accessed for successive accesses [82] for a smaller power-of-two number of banks. Power-of-two strides can occur, for example, in multi-dimensional arrays with power-of-two dimensions. To avoid going to the same bank in successive accesses, the usual solution used is a hashing scheme in hardware [82, 83, 84] that maps addresses to banks. The hash function is selected to behave like a pseudo-random function, such that, for most constant strides (including powers of two), successive accesses map to different banks.

Compilers for vector machines, such as the one by Corinna Lee [81], have the task of identifying cases where vector instructions are applicable, and generating code that uses vector instructions. This compiler task is called vectorization. Lee identifies four properties required of a code fragment for it to be vectorizable: first, it must be a loop; second, it must have at least one array variable; third, the addresses accessed by each array reference must form an arithmetic progression; and fourth, any statement containing an array reference cannot depend on itself. Lee goes on to present a vector-instruction scheduling algorithm, and a register-assignment algorithm.

Bank disambiguation and vectorizing compilers solve different problems. Software-exposed architectures require the compiler to know the actual banks accessed for each reference, not just that the banks are different as in vector machines. The bank disambiguation methods in this thesis provide bank numbers, while vectorizing compilers have no need to. Bank disambiguation enables avoiding arbitration logic, and long wires from processing elements to memory, as explained in chapter 1[4]. The major memory-parallelism requirement for vector machines is that successive accesses in a vector-memory instruction go to different banks; a requirement that is satisfied by hardware hashing schemes for any constant stride. Recognition of vector-memory instructions is done using affine-specific techniques, conceptually similar to modulo unrolling. Modulo unrolling, however, provides the additional guarantee that the banks are known, not just different; a property useful for bank-exposed architectures. Krste Asanovic (MIT) pointed out an interesting possibility of using modulo unrolling information in vectorizing compilers. In vectorizable loops, there are often more than one vector-memory instructions ready to issue. The issue-order selected impacts performance: if instructions that conflict in the banks accessed are issued together, then performance suffers compared to the case when non-conflicting instructions are issued together. For vector machines, the pre-conditioning loop produced by modulo unrolling can be used to ensure that the main loop begins with a known bank number for each affine reference. The main loop is then vectorized, not unrolled. The known starting bank number for each vector-memory instruction in the main loop can be used to determine an issue-order that minimizes bank conflicts.

Comparing the relative strengths of vector machines and bank-exposed architectures is difficult, as they have different motivations, problem domains and design elements. Nevertheless, we attempt a brief overview. Two advantages of vector machines are as follows. First, vector machines have smaller I-cache traffic than bank-exposed architectures, superscalars or VLIWs: vector machines encode many operations in a single instruction, leading to very compact code. Second, vectorizing compiler technology is mature, and does not need to perform bank disambiguation for good performance. Four advantages of bank-exposed architectures over vector machines are as follows. First, vector machines perform well primarily on affine-function scientific programs; bank-exposed machines can profit from programs with non-affine accesses as well. In particular, vectorizing compilers cannot exploit the non-affine memory parallelism from equivalence-class unification any

[4] Arbitration logic and wire-delay concerns are less important for vector machines. Since latency for main memory access is usually dozens of cycles, the added fractional cost of arbitration logic is low; for fast caches, however, arbitration logic increases latency by a larger fraction. Similarly, wire-delay adds a larger fraction to cache access time than to main-memory access time.

better than a conventional microprocessor. Second, modulo unrolling tolerates non-affine accesses in its loops along with affine array accesses, finding available parallelism from each class of access. In vector machines, non-affine accesses prevent vectorization if they have scalar or memory dependences with the affine accesses. Third, vector machine characteristics do not allow loops with self-dependent array accesses to be vectorized [81], while bank-exposed machines have no such restriction. Fourth, vector machines primarily exploit parallelism across different vector instructions, not within vector instructions. Bank-exposed architectures exploit parallelism within a single vectorizable group of operations, as well as across different vectorizable groups.

10.6 Compilers for systolic arrays

Systolic array architectures, such as Warp [85] and iWarp [17], consist of a array of processors, usually connected in a linear array or a 2-dimensional mesh, that communicate with each other using uni-directional or bi-directional links. Communication is directly CPU-to-CPU at the register-level rather than through memory; [17] argues the advantages of register communication in terms of reducing the number of memory accesses, and reducing size of memory required, as many temporary values need never to be stored. One similarity of systolic architectures with the Raw machine is the use of an exposed register-level communication network between the different processing elements.

Compilers for systolic arrays, such as the one by Ribas [86] for the Warp machine [85] and other systolic compilers [87, 88], have demonstrated automatic conversion of dense-matrix scientific codes into parallel systolic programs. Ribas exploits pipelined parallelism by mapping dependence chains onto systolic communication channels for programs with affine-function accesses. Mapping dependence chains to a systolic array is profitable when long dependence chains are available and re-used for different data elements. Re-use of the dependence-chains is key; otherwise, no pipelined parallelism is available. Ribas detects the dependence chains using his own dependence analysis technique, which formulates finding dependence as a linear integer programming problem. The resulting dependence patterns are represented as dependence vectors. Ribas shows that dependences can be mapped to systolic arrays when the dependence vectors are constant. For some affine-function codes, the dependence vectors are not constant; Ribas shows additional program transformations, that for some of these non-constant cases, result in constant dependence vectors after transformation.

Systolic compilers tackle a different problem from Maps: while systolic compilers deal with decomposing computation into pipelines, Maps performs bank disambiguation in an ILP-exploiting framework. Systolic mappings do not distribute their primary data among different systolic cells; instead the input data is streamed in from one end of the systolic array and output data is produced at the other. Memory has no fixed home bank on the systolic array; hence the question of disambiguation does not arise. In terms of the systolic computation model itself, it is profitable only for programs with long, regular and re-used dependence chains. The ILP-centric approach in our compiler is more general, giving comparable speedups to systolic arrays for systolic programs, while successfully parallelizing others as well. The Raw machine itself can be programmed as a systolic array if desired, however, it is not restricted to that programming model.

While the designers of systolic array machines like Warp [85] and iWarp [17] intended them to be used with a systolic compilation model, it is conceivable to apply an ILP-centric model to them, like that used in Maps. In particular, IWarp shares many features with the MIT Raw machine, such as exposed instruction-issue slots and exposed memory banks, and a compiler-routed communication network. Both are organized as tiles on a 2-D mesh, with each tile containing processing and memory. While we have not done a detailed study, it seems that Maps is applicable to Iwarp in principle. Iwarp satisfies the two prerequisites for Maps to apply: a bank-exposed architecture, and a network that provides compiler-ordering of messages. One source of possible performance loss is that Iwarp's network is optimized for streaming data and not the scalar communication required by Maps; in particular, it takes 4 cycles to set up a communication channel, but a 1 cycle throughput is possible. A lower setup time would help performance of Maps on Iwarp.

Chapter 11

Conclusion

This thesis presents a way improve the memory parallelism of microprocessor designs by using a compiler-centric approach. Most microprocessors today present a unified view of memory using hardware; unfortunately such hardware is difficult to scale to a high degree of memory parallelism. Two factors inhibit scalability: all-to-all dependence checks required by unified memory hardware, and the use of long wires in such hardware. Consequently, conventional microprocessors exploit only a limited amount of memory parallelism, usually no more than 1 or 2 memory instructions per cycle. This thesis explores an alternative approach: eliminate the unified memory hardware, and instead have the compiler provide a unified view of memory to the programmer. Eliminating unified memory hardware implies that such architectures have multiple memory banks visible to the software. These architectures are called *bank-exposed architectures*. The programmer continues to program in a convenient sequential model; the compiler automatically converts sequential programs to programs targeting many exposed banks. Memory parallelism is thus improved without increase in programming effort or memory latency.

This thesis shows that *bank disambiguation* is the key compiler technology required for good performance on bank-exposed architectures. Bank disambiguation is the act of ensuring that a memory reference instruction goes to a single compile-time-known bank in every dynamic instance. For most data distributions, bank disambiguation is not even possible for a majority of the references, as references may access different banks during different dynamic instances. Successful bank disambiguation schemes use carefully selected data distributions such that most references access a single bank, yet a high degree of memory parallelism exists between different references. Experimental results demonstrate that using the bank disambiguation methods in this thesis improves performance, in many cases, by a factor of 3 to 5 over not using bank disambiguation.

Bank disambiguation provides performance advantages as well as power advantages. A performance advantage in gained in two ways. First, since disambiguated references do not need to go through unified memory hardware, they avoid its non-scalable delay, thus making references faster, and allowing the number of banks to be increased. Second, since disambiguated references access known banks, the compiler can place instructions close to data, thus minimizing the distance traveled on chip to memory banks. This reduction in wire delay will become increasingly important over the next decade as wire-delay across chip, measured in cycles, is projected to rapidly rise. Besides performance, bank disambiguation also reduces power consumption. Power savings are realized for two reasons. First, not using complex, unified memory hardware implies that no power is used in such hardware. Second, shorter distances traveled on chip by memory references reduces power dissipation from resistance.

While bank disambiguation applies to any bank-exposed architecture, two domains are identified: general-purpose microprocessors and digital-signal processing (DSP) chips. The evaluation in this thesis focuses on general-purpose machines. Such machines have rapidly improved in performance over the last two decades; however, future performance gains are threatened by non-scalable memory parallelism and increasing wire delay on chip, measured in number of cycles. Billion-transistor architecture proposals have focused on compiler-reliant schemes that overcome these challenges through compiler knowledge. Bank disambiguation is an exciting technology for this era; bank disambiguation can minimize wire delay, and at the same time help scale memory parallelism to a high degree.

DSP chips, on the other hand, have for long used multiple exposed memory banks. DSP chips are used in many embedded systems, including portable devices; embedded system chips have recently exceeded general-purpose processors in dollar volume. For DSP chips, the lack of unified memory hardware translates to lower cost and lower power consumption. Low cost and power are particularly important to the DSP domain. Unfortunately, compiler technology for such chips has lagged behind their usage – even today, most DSP chips are hand-programmed in assembly language. The bank disambiguation methods in this thesis are a step towards automating this process. Exciting future work in bank disambiguation could focus on applying the methods in this thesis to the DSP domain.

Maps presents two disambiguation methods to ensure that most references satisfy the disambiguation property. First, equivalence class unification targets all programs, even those with irregular memory references such as references using unrestricted pointers, structures, pointer-based data structures and heap-allocated objects in their addresses. The quality of disambiguation can vary with the structure of the program. The second method, modulo unrolling, is an optimization for arrays whose references are primarily affine function accesses. Dense-matrix scientific codes and some multimedia codes benefit from modulo unrolling. Modulo unrolling also improved the performance of some of our irregular applications. The reason is that most programs contain some affine memory accesses, and modulo unrolling remains effective for these programs even in the presence of irregular references in the same basic block.

Bank disambiguation methods yielded most of the performance gains demonstrated in this thesis. Using equivalence-class unification, application performance improved by a factor of two over the speedups obtained using ILP alone. This improvement is fairly consistent over all classes of applications. Modulo unrolling is able to boost the performance of dense-matrix codes and some multimedia codes further, often to a factor of three to five over using ILP alone. The overall speedups on these codes was competitive with their speedup on multiprocessors, even though our approach exploited ILP rather than coarse-grained parallelism. Modulo unrolling gave some improvements in irregular programs for reasons given above.

Further, we show that selective use of non-disambiguated (dynamic) references is helpful in certain cases to augment bank disambiguation. One benefit of dynamic accesses is when dynamic support allows arrays with non-affine accesses to be distributed, possibly exposing more memory parallelism and attaining better speedups. Another benefit of dynamic accesses is to use them for infrequent irregular references to arrays, allowing more frequently accessed portions to be bank disambiguated via modulo unrolling. Finally using dynamic accesses for a few 'bad' references can prevent excessive merging of equivalence classes, yielding higher memory parallelism. Software serial ordering is introduced as an efficient method of enforcing dependences between dynamic accesses. Optimizations on software serial ordering, called independent epochs and updates, are also presented.

We are encouraged by the results of the Maps approach to memory orchestration. Maps is able to exploit memory parallelism in a range of applications, from those containing small amounts of memory parallelism to more regular applications with large amounts of memory parallelism. This versatility opens up a range of possible applications for Maps. From small embedded designs to desktop microprocessor-based systems to supercomputers, machines with exposed memory banks will benefit from our techniques.

Bibliography

[1] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman Amarasinghe, and Anant Agarwal. Baring It All to Software: Raw Machines. *IEEE Computer*, 30(9):86–93, September 1997. Also available as MIT-LCS-TR-709.

[2] Jonathan Babb, Matthew Frank, Victor Lee, Elliot Waingold, Rajeev Barua, Michael Taylor, Jang Kim, Srikrishna Devabhaktuni, and Anant Agarwal. The raw benchmark suite: Computation structures for general purpose computing. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, CA, April 1997.

[3] Dick Sites. It's the Memory, Stupid! *Microprocessor Report*, 10(10):19, August 5 1996.

[4] Tom Burd. *http://infopad.eecs.berkeley.edu/CIC/summary/local/*. 1999. Table derived from website and related sources.

[5] Linley Gwennap. Alpha 21364 to Ease Memory Bottleneck. *Microprocessor Report*, 12(14):12, October 26 1998.

[6] Linley Gwennap. Digital 21264 Sets New Standard. *Microprocessor Report*, 10(14):11, October 28 1996.

[7] Doug Matzke. Will Physical Scalability Sabotage Performance Gains? *Computer*, pages 37–39, September 1997.

[8] International Technology Roadmap for Semiconductors, 1998 Update. *Semiconductor Industry Association*, page 4, 1998.

[9] F. Thomson Leighton. *Introduction to parallel algorithms and architectures: arrays, trees, hypercubes.* Morgan Kaufmann, San Mateo, CA, 1992.

[10] William Dally. Memory-Centric Computers. *Microprocessor Report*, 10(10):21, August 5 1996.

[11] Peter Song. Demystifying EPIC and IA-64. *Microprocessor Report*, 12(1):21, January 26 1998.

[12] Linley Gwennap. IA-64: A Parallel Instruction Set. *Microprocessor Report*, 13(7):1, May 31 1999.

[13] Rich Belgard. Transmeta Exposed. *Microprocessor Report*, 12(16):9, December 7 1998.

[14] Jim Turley. Tensilica CPU Bends to Designers' Will. *Microprocessor Report*, 13(3):12, March 8 1999.

[15] Tom R. Halfhill. Sun Reveals secrets of "Magic". *Microprocessor Report*, 13(11):13, August 23 1999.

[16] Joseph A. Fisher. Very Long Instruction Word Architectures and the ELI-512. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, pages 140–150, Stockholm, Sweden, June 1983.

[17] S. Borkar, R. Cohn, G. Cox, T. Gross, H. T. Kung, M. Lam, M. Levine, B. Moore, W. Moore, C. Peterson, J. Susman, J. Sutton, J. Urbanski, and J. Webb. Supporting Systolic and Memory Communication in iWarp. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 70–81, June 1990.

[18] *DSP56000 24-bit Digital Signal Processor Family Manual.* Motorola, 1995. Also available at www.motorola.com.

[19] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture (ISCA) 1994*, Chicago, IL, April 1994. IEEE.

[20] Anant Agarwal, David Chaiken, Godfrey D'Souza, Kirk Johnson, David Kranz, John Kubiatowicz, Kiyoshi Kurihara, Beng-Hong Lim, Gino Maa, Dan Nussbaum, Mike Parkin, and Donald Yeung. The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor. In *Proceedings of Workshop on Scalable Shared Memory Multiprocessors*. Kluwer Academic Publishers, 1991. An extended version of this paper appears as MIT/LCS Memo TM-454, 1991.

[21] Tilak Agerwala, Joanne Martin, Jamshed H. Mirza, David Sadler, Dan Dias, and Marc Snir. SP2 System Architecture. *IBM Corporation Research Report IBM-RC-20012*, Jan 1995.

[22] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam. Maximizing Multiprocessor Performance with the SUIF Compiler. *IEEE Computer*, 29(12):84–89, December 1996.

[23] William Blume, Rudolf Eigenmann, Keith Faigin, John Grout, Jay Hoeflinger, David Padua, Paul Petersen, Bill Pottenger, Lawrence Rauchwerger, Peng Tu, and Stephen Weatherford. Effective Automatic Parallelization with Polaris. *International Journal of Parallel Programming*, May 1995.

[24] Robert Wilson, Robert French, Christopher Wilson, Saman Amarasinghe, Jennifer Anderson, Steve Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary Hall, Monica Lam, and John Hennessy. SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers. *ACM SIGPLAN Notices*, 29(12), December 1996.

[25] Walter Lee, Rajeev Barua, Matthew Frank, Devabhatuni Srikrishna, Jonathan Babb, Vivek Sarkar, and Saman Amarasinghe. Space-Time Scheduling of Instruction-Level Parallelism on a Raw Machine. In *Proceedings of the Eighth ACM Conference on Architectural Support for Programming Languages and Operating Systems*, pages 46–57, San Jose, CA, October 1998.

[26] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, CA, 1997.

[27] P. Paulin, C. Liem, M. Cornero, F. Nacabal, and G. Goossens. Embedded Software in Real-Time Signal Processing Systems: Application and Architecture Trends. *Invited paper, Proceedings of the IEEE*, 85(3), March 1997.

[28] S. Przybylski, T. Gross, J. Hennessy, N. Jouppi, and C. Rowen. Organization and VLSI Implementation of MIPS. *Journal of VLSI and Computer Systems*, 1(2):170–208, December 1984.

[29] Keith Diefendorff. K7 Challenges Intel. *Microprocessor Report*, 12(14):1, October 26 1998.

[30] John Hennessy, Norman Jouppi, Forest Baskett, and John Gill. Mips: A vlsi processor architecture. Technical Report 223, Stanford University, Stanford, CA, June 1983.

[31] Robert P. Colwell, Robert P. Nix, John J. O'Donnell, David P. Papworth, and Paul K. Rodman. A VLIW architecture for a trace scheduling compiler. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 180–192, Palo Alto, California, October 5–8, 1987.

[32] John Hennessy and David Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann, Palo Alto, CA, second edition, 1996.

[33] Linley Gwennap. AMD Gets the IIIrd Degree. *Microprocessor Report*, 13(3):22, March 8 1999.

[34] Michael B. Taylor. Design Decisions in the Implementation of a Raw Architecture Workstation. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, September 1999.

[35] R. Rugina and M. Rinard. Pointer Analysis for Multithreaded Programs. In *Proceedings of the SIGPLAN '99 Conference on Program Language Design and Implementation*, Atlanta, GA, May 1999.

[36] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Languages Design and Implementation*, Orlando, FL, June 1994.

[37] D. E. Maydan, J. L. Hennessy, and M. S. Lam. Efficient and Exact Data Dependence Analysis. In *Proceedings of the SIGPLAN '91 Conference on Program Language Design and Implementation*, Toronto, June 1991.

[38] Saman Amarasinghe, Walter Lee, and Ben Greenwald. Strength Reduction of Integer Division and Modulo Operations. *MIT Laboratory for Computer Science Technical Memo, MIT-LCS-TM-600*, November 1999.

[39] Saman Amarasinghe. Parallelizing Compiler Techniques Based on Linear Inequalities. In *Ph.D Thesis, Stanford University. Also appears as Techical Report CSL-TR-97-714*, January 1997.

[40] Anant Agarwal, David Kranz, and Venkat Natarajan. Automatic Partitioning of Parallel Loops for Cache-Coherent Multiprocessors. In *22nd International Conference on Parallel Processing*, pages 943–962, St. Charles, IL, August 1993. IEEE. Also in IEEE Transactions on Parallel and Distributed Systems, vol 6, pp 943-961, September 1995.

[41] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *toplas*, 1987.

[42] Michael D. Smith. Extending SUIF for Machine-dependent Optimizations. In *Proceedings of the First SUIF Compiler Workshop*, pages 14–25, Stanford, CA, January 1996.

[43] Bjarne Steensgaard. Points-to analysis in almost linear time. St. Petersburg Beach, FL, January 1996.

[44] Robert P. Wilson. Efficient Context-Sensitive Pointer Analysis For C Programs. In *Ph.D Thesis, Stanford University, Computer Systems Laboratory*, December 1997.

[45] Standard Performance Evaluation Corporation. The SPEC benchmark suites. *http://www.spec.org/*.

[46] Martin C. Rinard and Monica S. Lam. The Design, Implementation, and Evaluation of Jade. *ACM Transactions on Programming Languages and Systems*, 20(3):483–545, May 1998.

[47] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In *Proceedings of the 30th Annual International Symposium on Microarchitecture (MICRO-30)*, Research Triangle Park, NC, December 1997. IEEE Computer Society.

[48] Lawrence A. Rowe, Kevin Gong, Eugene Hung, Ketan Patel, Steve Smoot, and Dan Wallach. Berkeley MPEG Tools. *http://bmrc.berkeley.edu/frame/research/mpeg/*.

[49] Corinna Lee and Mark Stoodley. UTDSP BenchMark Suite. *http://www.eecg.toronto.edu/ corinna/DSP/infrastructure/UTDSP.html*, 1992.

[50] Joel Saltz, Ravi Ponnusamy, Shamik Sharma, Bongki Moon, Yuan-Shin Hwang, Mustafa Uysal, and Raja Das. A Manual for the CHAOS Runtime Library. Technical report, University of Maryland: Department of Computer Science and UMIACS, March 1995.

[51] Bruce Schneier. *Applied cryptography: protocols, algorithms, and source code in C.* John Wiley, New York, second edition, 1996.

[52] International Standards Organization (ISO). MPEG-1 Video standard. *ISO CD 11172-2*. Also at http://www.iso.ch/.

[53] Keith D. Cooper and Timothy J. Harvey. Compiler-controlled memory. In *Proceedings of the Eighth ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, pages 2–11, San Jose, CA, October 1998.

[54] Mazen A. R. Saghir, Paul Chow, and Corinna G. Lee. Exploiting dual data-memory banks in digital signal processors. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 234–243, Cambridge, MA, October 1–5, 1996.

[55] *NEC μPD7701x Family User's Guide.* NEC Corporation, 1995.

[56] Ashok Sudarsanam and Sharad Malik. Memory Bank and Register Allocation in Software Synthesis for ASIPs. *Proceedings of the International Conference on Computer-Aided Design*, pages 388–392, 1995.

[57] D. B. Powell, E. A. Lee, and W. C. Newman. Direct Synthesis of Optimized DSP Assembly Code from Signal Flow Block Diagrams. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, volume 5, pages 553–556, 1992.

[58] B. Wess. Automatic Code Generation for Integrated Digital Signal Processors. In *Proceedings of the International Symposium on Circuits and Systems*, pages 33–36, 1991.

[59] P. Lowney, S. Freudenberger, T. Karzes, W. Lichtenstein, R. Nix, J. O'Donnell, and J. Ruttenberg. The Multiflow Trace Scheduling Compiler. In *Journal of Supercomputing*, pages 51–142, January 1993.

[60] Alexandru Nicolau. Run-Time Disambiguation: Coping with Statically Unpredictable Dependences. *IEEE Transactions on Computers*, 38(2), May 1989.

[61] William Y. Chen. *Data Preload for Superscalar and VLIW processors.* PhD thesis, Dept. of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, Urbana, IL, 1993.

[62] David M. Gallagher, William Y. Chen, Scott A. Mahlke, John C. Gyllenhaal, and Wen mei W. Hwu. Dynamic Memory Disambiguation Using the Memory Conflict Buffer. In *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, San Jose, CA, October 1994. ACM.

[63] David M. Gallagher. *Memory Disambiguation to Facilitate Instruction-Level Parallelism Compilation.* PhD thesis, Dept. of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, Urbana, IL, 1995.

[64] Dror E. Maydan. Accurate Analysis of Array References. In *Ph.D Thesis, Stanford University. Also appears as Techical Reports CSL-TR-92-547, STAN-CS-92-1449*, September 1992.

[65] U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, Boston, 1988.

[66] Randy Allen and Ken Kennedy. Automatic translation of fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[67] M. Wolfe. *Optimizing Supercompilers for Supercomputers.* The MIT Press, Cambridge, MA, 1989.

[68] Markus Willems, Holger Keding, Vojin Zivojnovic, and Heinrich Meyr. Modulo-Addressing Utilization in Automatic Software Synthesis for Digital Signal Processors. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, pages 687–690, 1997.

[69] Vojin Zivojnovic. Compilers for Digital Signal Processors: The Hard Way from Marketing- to Production-Tool. *DSP and Multimedia Technology Magazine*, 4(5):27–45, July/August 1995.

[70] Raja Das, Mustafa Uysal, Joel Saltz, and Yuan-Shin Hwang. Communication Optimizations for Irregular Scientific Computations on Distributed Memory Architectures. *Journal of Parallel and Distributed Computing*, 22(3), September 1994.

[71] Shubhendu Mukherjee, Shamik Sharma, Mark Hill, James Larus, Anne Rogers, and Joel Saltz. Efficient Support for Irregular Applications on Distributed-Memory Machines. In *Principles and Practice of Parallel Programming (PPoPP) 1995*, pages 68–79, Santa Clara, CA, July 1995. ACM.

[72] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiling Fortran D for MIMD Distributed Memory Machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[73] Saman P. Amarasinghe and Monica S. Lam. Communication Optimization and Code Generation for Distributed Memory Machines. In *Proceedings of SIGPLAN '93, Conference on Programming Languages Design and Implementation*, June 1993.

[74] R. Barua, D. Kranz, and A. Agarwal. Communication-Minimal Partitioning of Parallel Loops and Data Arrays for Cache-Coherent Distributed-Memory Multiprocessors. In *Languages and Compilters for Parallel Computing*, pages 350–368. Springer-Verlag Publishers, August 1996.

[75] J. Ramanujam and P. Sadayappan. Compile-Time Techniques for Data Distribution in Distributed Memory Machines. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):472–482, October 1991.

[76] Jennifer M. Anderson and Monica S. Lam. Global Optimizations for Parallelism and Locality on Scalable Parallel Machines. In *Proceedings of SIGPLAN '93 Conference on Programming Languages Design and Implementation.* ACM, June 1993.

[77] Manish Gupta and Prithviraj Banerjee. PARADIGM: A Compiler for Automatic Data Distribution on Multicomputers. In *Proceedings 1993 International Conference on Supercomputing*, Tokyo, Japan, July 1993. ACM.

[78] Sandhya Dwarkadas, Alan L. Cox, and Willy Zwaenepoel. An Integrated Compile-Time/Run-Time Software Distributed Shared Memory System. In *Proceedings of the*

Seventh International Conference on Architectural Support for Programming Languages and Operating Systems, pages 186–197, Cambridge, MA, October 1–5, 1996.

[79] Daniel J. Scales, Kourosh Gharachorloo, and Chandramohan A. Thekkath. Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 174–185, Cambridge, Massachusetts, October 1–5, 1996.

[80] Krste Asanovic. *Vector Microprocessors.* PhD thesis, University of California, Berkeley, May 1998.

[81] Corinna G. Lee. *Code Optimizers and Register Organizations for Vector Architectures.* PhD thesis, University of California, Berkeley, May 1992.

[82] B. Ramakrishna Rau. Pseudo-Randomly Interleaved Memory. In *Proceedings of the 18rd International Symposium on Computer Architecture (ISCA)*, volume 19, pages 74–83, May 1991.

[83] Mayez A. Al-Mouhamed and Steven S. Seiden. Minimization of memory and network contention for accessing arbitrary data patterns in simd systems. *IEEE Transactions on Computers*, 45(6):757–762, June 1996.

[84] M. Valero, T. Lang, J. M. Llaberia, M. Peiron, E. Ayguade, and J. J. Navarro. Increasing the Number of Strides for Conflict-free Vector Access. In *Proceedings of the 19th International Symposium on Computer Architecture*, Gold Coast, Australia, May 1992.

[85] M. Annaratone et al. Warp architecture and implementation. In *Proc. 13th Annual International Symposium on Computer Architecture*, pages 346–356, Los Alamitos, Calif., 1986. IEEE Computer Society Press.

[86] Hudson B. Ribas. *Automatic Generation of Systolic Programs from Nested Loops.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, June 1990.

[87] R. Capello, Peter Engstrom, and Bradley R. Engstrom. The Sdef Systolic Programming System. In *Proceedings International Conference on Parallel Processing (ICPP)*, pages 645–652. ACM, 1987.

[88] Sanjay V. Rajopadhye, S. Purushothaman, and Richard M. Fujimoto. On Synthesizing Systolic Arrays from Recurrence Equations with Linear Dependencies. In *Proceedings of the Sixth Conference on Foundations of Software Technology and Theoretical Computer Science*, New Delhi, India, December 1986. Springer Verlag, LNCS No. 241.