Compiler Support for Scalable and Efficient Memory Systems

Rajeev Barua† , Walter Lee‡ , Saman Amarasinghe‡ , Anant Agarwal‡ *

ECE Department, University of Maryland†
College Park, MD 20742, U.S.A
barua@eng,umd.edu
http://www.ece.umd.edu/~barua

M.I.T. Laboratory for Computer Science‡
Cambridge, MA 02139, U.S.A.
{walt,saman,agarwal}@lcs.mit.edu
http://www.cag.lcs.mit.edu/raw

Abstract

Technological trends require that future scalable microprocessors be decentralized. Applying these trends toward memory systems shows that size of cache accessible in a single cycle will decrease in future generation of chips. Thus, a *bank-exposed memory system* comprising of small, decentralized cache banks must eventually replace that of a monolithic cache. This paper considers how to effectively use such a memory system for sequential programs.

This paper presents Maps, the software technology central to *bank-exposed architectures*, which are architectures with bank-exposed memory systems. Maps solves the problem of *bank disambiguation* – that of determining at compile-time which bank a memory reference is accessing. Bank disambiguation is important because it enables the compile-time optimization for data locality, where data can be placed closed to the computation that requires it. Two methods for bank disambiguation are presented: *equivalence-class unification* and *modulo unrolling*. Experimental results are presented using a compiler for the MIT Raw machine, a bank-exposed architecture that relies on the compiler to (1) manage its memory and (2) orchestrate its instruction level parallelism and communication. Results on Raw using sequential codes demonstrate that using bank disambiguation improves performance by a factor of 3 to 5 over using ILP alone.

1 Introduction

Technological trends are forcing computer architects to reexamine their assumptions. In future technology, wire delay is scaling approximately with feature size, but a combination of decreasing transistor size and increasing die size means that the fraction of the chip

*This research is funded by Darpa contract # DABT63-96-C-0036. Walter Lee is funded in part by an IBM Research Fellowship.

reachable by a wire in a single cycle is diminishing dramatically [1, 14, 31]. Even with conservative clock scaling, this fraction is projected to reduce to less than 2% in the next decade [1]. Consequently, design complexity will be bound not by chip capacity but by wire latency. Future microprocessors must be built out of localized structures that do not require long wires.

Modern superscalars are full of centralized components that do not scale, such as issue logic, register window, wakeup logic, and bypass paths. Some have proposed to addressed the scalability problem incrementally, by identifying the least scalable components and proposing more decentralized designs of those components. This approach, however, only addresses the scalability problem in the short term. Wire latency will prevent a design with just one single point of centralization from continuing to take advantage of the exponential increasing in transistor budget. For a microprocessor to be scalable, its entire processing core, from the dispatch unit to the register file to the functional units, must be decentralized. A natural way to achieve this goal is to design future microprocessors by replicating technologically feasible processing cores.

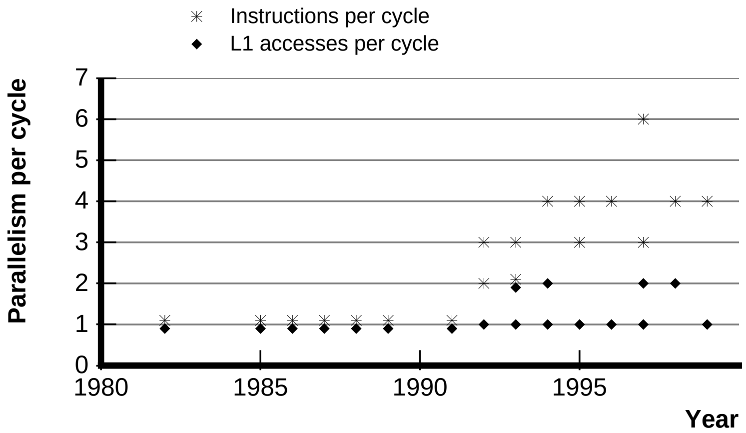

Figure 1: ILP and memory parallelism for various microprocessors over the last two decades. Points in the graph represent commercial microprocessors from that year. Each microprocessor contributes two points, one for its maximum number of instructions per cycle (ILP), and one for its maximum number of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

A truly scalable microprocessor, however, not only entails a scalable processing core – the memory system must scale as well. A recent study shows that cache access time in terms of gate delay scales poorly with capacity and sub-linearly with technology [1]. Thus, the size of cache accessible in a single cycle will decrease in future generation of chips. For example, in the aggressive 2014 technology projected by the SIA technology roadmap [25], a 512-byte cache will require three cycles to access. Furthermore, as shown in Figure 1, the number of cache ports has not scaled with the number of functional units in a microprocessor.

Based on these observations, it is no longer reasonable to design the memory system in a future microprocessor as a single monolithic unit. A conventional hierarchical memory system does not address the issues, because data from the centralized L1 cache may still traverse a potentially large distance to reach the growing number of compute elements that can fit on a chip. To maintain scalability, the memory system must be decentralized.

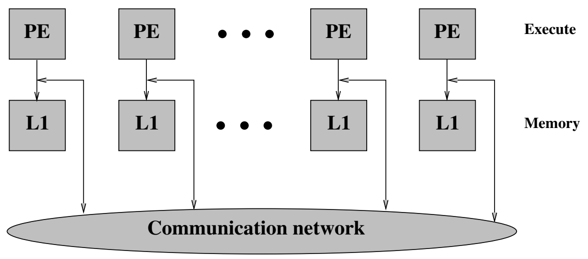

Figure 2: A bank-exposed architecture comprises units of processing element and a memory bank. Each processing element is tightly coupled with a memory bank, but it can access all the memory banks through a communication network.

Figure 2 shows an abstract view of a scalable, fully decentralized microprocessor. The microprocessor comprises individual autonomous units, each with its own processing element and L1 cache bank. Each processing element has direct access to an L1 cache bank; a communication network can be used to access values in other cache banks. In this organization, cache banks are kept small, fast, and scalable with technology. Furthermore, because computation is mapped to each processing element statically, data can be statically placed near the processing element that needs it. We call such an architecture a *bank-exposed architecture*, because it allows the compiler to manage the locality of data along with computation.

This paper considers the challenge of utilizing a bank-exposed architecture for sequential programs. It focuses on *bank disambiguation*, a central problem in attaining good performance from such an architecture. Bank disambiguation is the problem of determining at compile-time which bank a memory reference is accessing. A particular load or store instruction is said to be *bank disambiguated* if the instruction accesses the same compile-time predictable bank every time it is executed. Bank disambiguation is important because it is a prerequisite for the compile-time optimization of data locality. When a memory reference is known to refer to memory on a particular bank, the computation that operates on it can be placed close to that bank.

This paper presents Maps, a compiler managed memory system that performs bank disambiguation for bank-exposed architecture. It presents two complimentary bank disambiguation techniques. *Equivalence class unification* uses pointer analysis to guide the intelligent placement of data. *Modulo unrolling* uses intelligent loop unrolling to turn undisambiguated access into disambiguated ones.

A good bank disambiguation scheme should satisfy three criteria. First, it should distribute data evenly across the memory banks. It is easy to bank disambiguate all accesses by mapping all data to a single bank, but that is inefficient use of memory and will likely lead to poor locality. Second, the distribution should lead to good locality, where data is close to the computation that uses it. Finally, when code transformation is involved, the scheme should try to minimize any increase in code size. The methods in this paper aim for balanced distribution and minimizes code growth. It provides the opportunity for a back end to optimize for locality, but it does not address the locality issue directly.

The methods in this paper apply to any bank-exposed architecture for both general-purpose and embedded systems [7, 11, 20, 22, 32]. In theory, it may also be used to map sequential programs onto distributed shared memory multiprocessors, although the communication latencies on DSMs have historically been too high to be able to profitably exploit the instruction level parallelism extracted by this compiler approach. This paper uses the Raw machine, a bank-exposed architecture, to illustrate its techniques.

The rest of this paper is organized as follows. Section 2 gives the background for our research. Sections 3 and 4 describe our two methods for bank disambiguation, equivalence-class unification and modulo unrolling, respectively. Section 5 presents experimental results. Section 6 describes related work. Section 7 concludes.

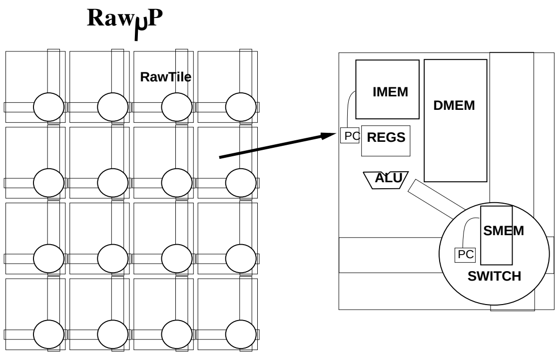

Figure 3: A Raw microprocessor is a mesh of tiles, each with a processing element, some memory and a switch. The processing element contains both registers and an ALU. The processing element interfaces with its local instruction memory, local data memory and local switch. The switch contains its own instruction memory.

2 Background

This section gives the background of our research. First, it describes the Raw architecture, the bank-exposed architecture on which we implement Maps. Then, it overviews the Raw compiler and its interface with Maps.

The Raw architecture Figure 3 depicts the Raw microprocessor. It 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 compiler-routed register-level network. Bank-disambiguated accesses to compile-time-known banks either complete over the static network or are local to a tile. The dynamic network is a slower runtime-routed network that serves the role of a conventional memory system's arbitration logic. Accesses to compile-time-unknown banks complete over the dynamic network. Each Raw tile has its own instruction stream; different tiles proceed in a loosely synchronous manner, communicating only for register and control dependences.

The Raw compiler Rawcc is the Raw parallelizing compiler based on the SUIF compiler infrastructure [33]. It takes a sequential C or Fortran program, extracts its instruction level parallelism, and parallelizes it across the tiles of the Raw machine. Figure 4 gives the major components of Rawcc.

Rawcc comprises two major components. Maps is the memory front end that performs

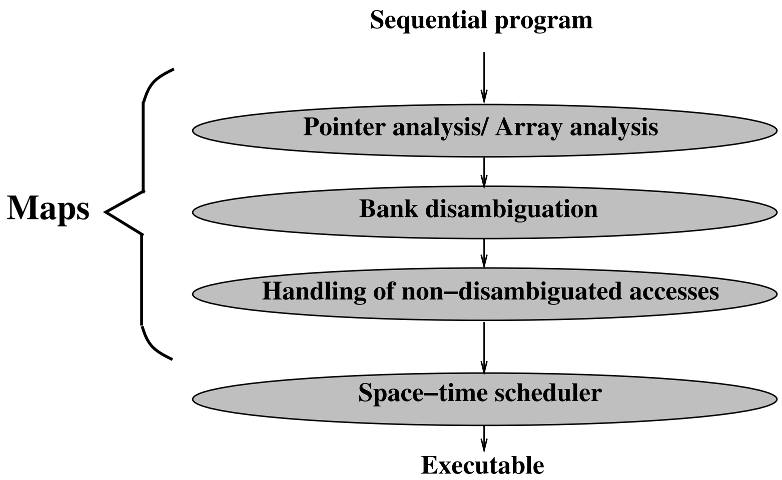

Figure 4: Structure of the Raw compiler.

bank disambiguation of memory accesses. It partitions all memory references and data objects into equivalent classes. Each equivalent class is labeled as either a single-tile equivalent class or a low-order-interleaved one. Objects in a single-tile equivalent class are mapped to a single tile. Objects in a low-order-interleaved class must be arrays; they are interleaved element-wise across the tiles. The Maps analysis ensures that memory references can be bank disambiguated if objects are mapped in such a manner.

Maps itself is made up of three components. It begins by collecting information from traditional pointer and array analysis. This information is then used to perform analysis for bank disambiguation. The third component deals with accesses that Maps decides not to bank disambiguate. This paper focuses on techniques for bank disambiguation; a description of handling for non-disambiguable access can be found in [5].

The space-time scheduler is the back end of Rawcc. It performs tasks related to the mapping of instruction level parallelism to the Raw tiles. In addition, it is responsible for mapping each equivalence class of data objects to the memory bank of a specific Raw tile. The space-time scheduler performs this mapping with two goals: it tries to map equivalence classes that are rarely concurrently accessed to the same physical bank, and it tries to map accesses close to the computation that needs it. See [17] for details.

3 Equivalence-class unification

This section describes equivalence-class unification, our first method for bank disambiguation. Equivalence class unification (ECU) is our baseline disambiguation technique. It is applicable to all memory accesses, including arbitrary array accesses, pointer dereferences, structure references, and heap references. ECU provides disambiguation through careful data placements that are guided by pointer analysis.

Section 3.1 gives an introduction to pointer analysis, while Section 3.2 describes ECU itself. Throughout this section, we use Figure 5 as an expository example.

3.1 Pointer analysis

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 [27], 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 5(a). Figure 5(b) shows the results of the SPAN pointer analysis package on the program. 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. An entire array is considered a single object, but each field in a structure is considered a separate object. Figure 5(b) shows the abstract data objects marked with *assign* comments, with the location set numbers for the objects listed alongside the comment. 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 reference can refer. In Figure 5(b), each memory reference 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 the compiler all objects are assigned such numbers. Dotted edges represent potential memory dependences derived from pointer analysis: two memory references are potentially dependent if 1. the intersection of their location-set lists is non-empty; 2. one of the accesses is a store.

3.2 Equivalence-class unification method

Figure 5 helps explain the ECU method through an example. First, ECU runs pointer analysis described above: Figure 5(b) shows the results of pointer analysis. Next, ECU represents the pointer analysis information as a bipartite graph of data objects and memory references. Figure 5(c) shows the bipartite graph for the program in Figure 5(b). A node is constructed for each abstract object and for each memory reference. The upper row shows the abstract objects, with the location-set number for each object in parentheses. The lower row shows the memory references. 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 *alias equivalence classes* from the bipartite 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. ECU derives the equivalence classes by computing the connected components of the bipartite graph. Figure 5(d) shows the bipartite graph in Figure 5(c) partitioned into four equivalence classes. References in the same alias class may 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. Figure 5(e) shows a sample mapping using the equivalence classes in Figure 5(d). Mapping of equivalence class to memory banks is performed by the space-time scheduler, the backend of the Raw compiler. The space-time scheduler performs this mapping with two goals: it tries to map virtual banks that are rarely concurrently accessed to the same physical bank, and it tries to map accesses close to the computation that needs it. See [17] for details.

Quality of the disambiguation The quality of the disambiguation from ECU depends upon the number and size of the alias classes. A large number of small classes gives the most memory parallelism, since accesses mapped to different classes can execute in parallel.

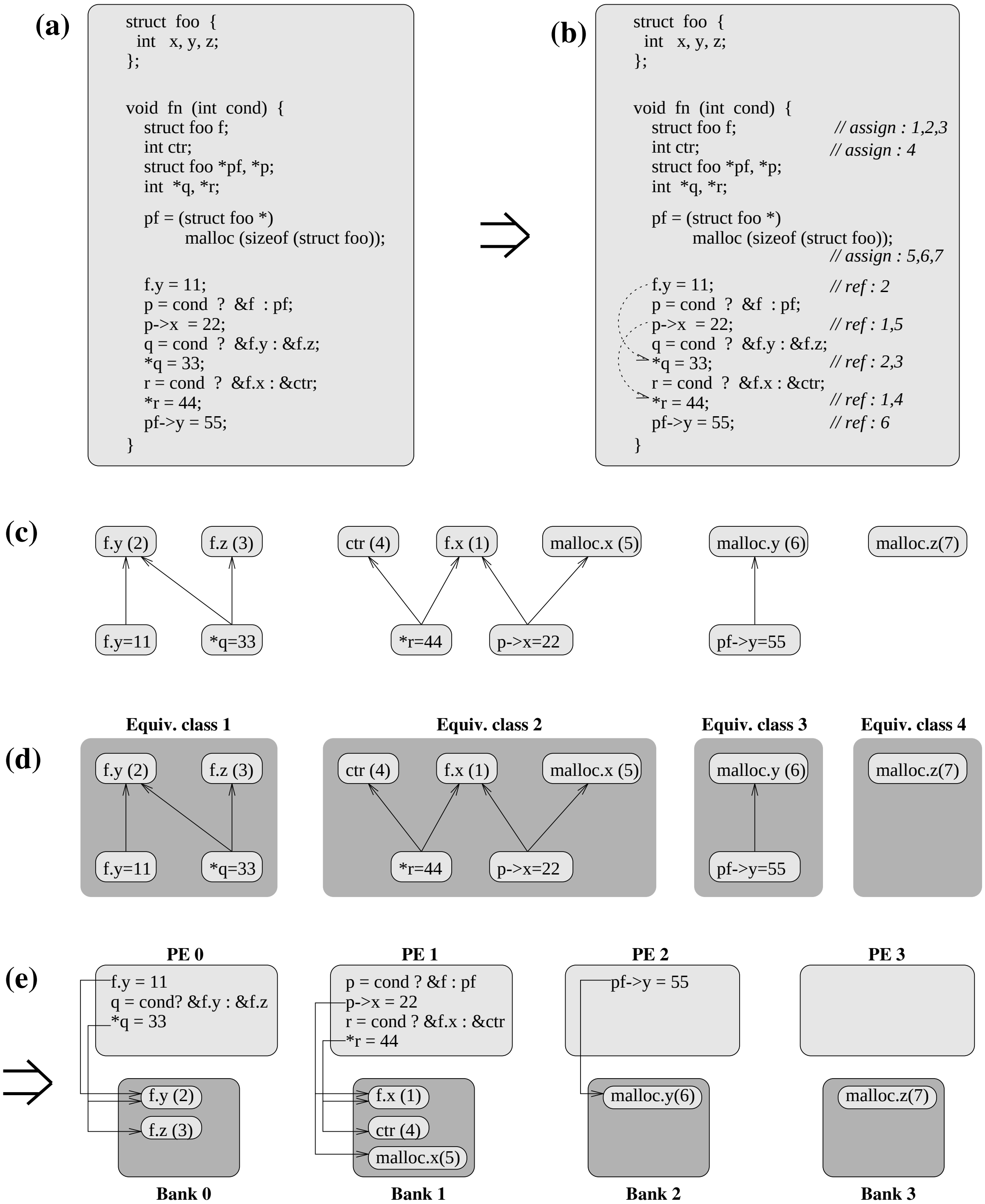

Figure 5: Example showing equivalence-class unification. (a) Initial program. (b) After pointer analysis, showing location-set numbers and dependence edges (dotted lines). (c) Memory objects and references represented as bipartite graph. (d) Connected components of bipartite graph marked as equivalence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

The number and size of the classes depend on the access patterns of the program, which the compiler cannot control. Nevertheless, our results in Section 5 show that many programs contain several alias classes. Finding any more than one class enables us to remove the bottleneck of a centralized memory system.

4 Modulo unrolling

The major limitation of equivalence-class unification is that an array belongs to one equivalence class and is mapped to only one bank. This section presents modulo unrolling, a technique for attaining bank disambiguation and memory parallelism for arrays. Modulo unrolling is applicable to array references whose index expressions are affine functions of enclosing loop induction variables.[1] Such accesses are common in dense-matrix scientific codes as well as some multimedia and streaming applications.

This section is organized as follows. Section 4.1 illustrates modulo unrolling through an example. Section 4.2 describes modulo unrolling and its scope. Section 4.3 proves that the unroll factor selected by modulo unrolling is necessary and sufficient. Section 4.4 discusses the issue of code growth.

4.1 Example

Figure 6 gives an example of modulo unrolling. Figure 6(a) shows the code fragment forming the compiler input, consisting of a simple for loop containing a single array reference $A[i]$. The array $A[]$ ranges from 0 to 99. To enable parallel accesses, the compiler distributes $A[]$ among 4 memory banks using low-order interleaving.[2] This distribution, however, makes $A[i]$ non-bank-disambiguable, because it touches data on all four banks.

In special cases, full unrolling can attain bank disambiguation. Figure 6(b) shows the

[1]An *affine function* of a set of variables is defined as a linear combination of those variables, plus a constant. *E.g:* 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.

[2]Low-order interleaving is the distribution of array elements in a round-robin manner across the memory banks. That is, for a low-order interleaved array $A[]$, element $A[i]$ is allocated on bank i mod N.

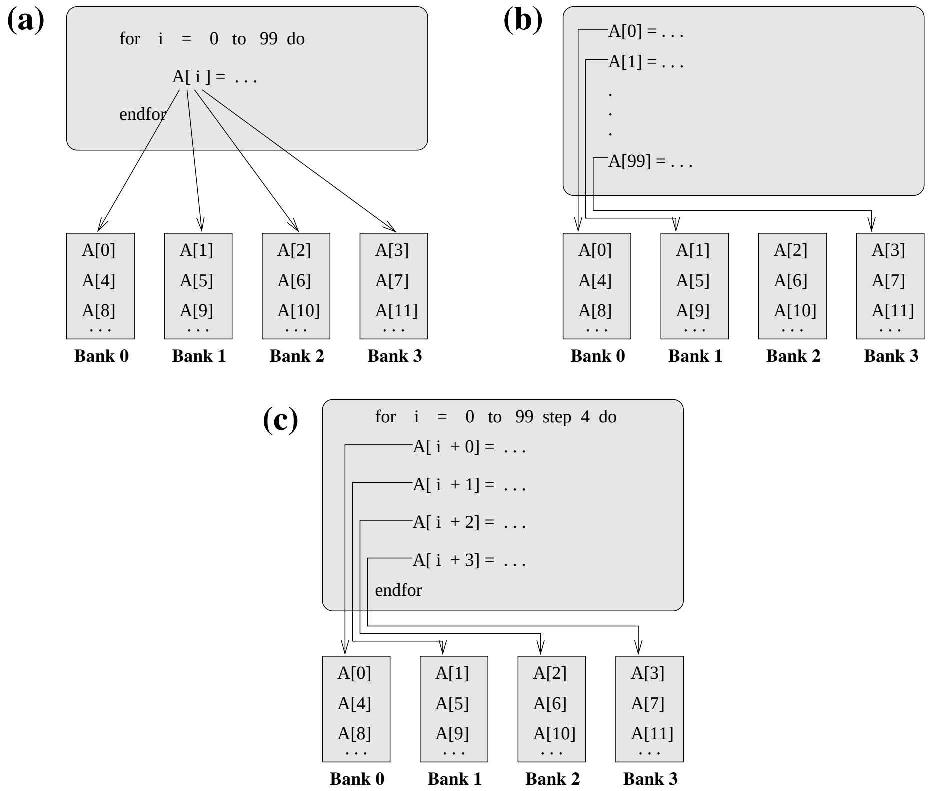

Figure 6: Example of modulo unrolling. (a) Original code. Array A is low-order interleaved on a 4-bank bank-exposed machine. The $A[i]$ reference instruction goes to different banks for different values of i. (b) Code after full unrolling. Disambiguation is attained, but code size is huge. (c) Code after unrolling by factor 4. Disambiguation is attained with limited code size increase.

sample loop in Figure 6(a) fully unrolled. This solution, however, is only possible if the loop bounds are known, and it is only reasonable if the iteration count is small.

Modulo unrolling uses a modest amount of intelligent unrolling to make the example access disambiguable. In the example, it unrolls the loop by a factor of four, as shown in Figure 6(c). After the unrolling, each access refers to elements on the same memory bank: $A[i]$ to tile 0, $A[i+1]$ to tile 1, $A[i+2]$ to tile 2, and $A[i+3]$ to tile 3. Thus, each access in the unrolled loop has become bank disambiguated. Furthermore, the accesses can proceed in parallel, thus providing memory parallelism.

4.2 Modulo unrolling method

Modulo unrolling is a technique for bank disambiguation that is applicable to array references whose index expressions are affine functions of enclosing loop induction variables. This section describes the method.

Modulo unrolling works as follows. First, the compiler looks for affine array accesses inside loop-nests. For each array access and each loop, it computes the minimum unroll factor required on the loop in order for the access to be bank disambiguated. Once the compiler computes the induced unroll factors for each loop for each affine access, the final unroll factor for a loop is the least common multiple (*lcm*) of all its unroll factors induced by each enclosing affine accesses. Section 4.3 proves that the overall code-growth from modulo unrolling is bounded by the number of memory banks in most cases, even for nested loops.

Let N be the number of memory banks in the target software-exposed architecture. We define the following:

Definition 4.1 *Given a k-dimensional (not necessarily perfectly nested) loop nest of the form:*

for $v_1 = l_1$ to u_1 step s_1
  for $v_2 = l_2$ to u_2 step s_2
    ....
    for $v_k = l_k$ to u_k step s_k
      `/* the loop body */`

We represent an affine access to an array A of dimension $MAX_1 \times MAX_2 \times \cdots \times MAX_d$ as follows:

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

Then, for each $j \in [1, k]$, the minimum unroll factor U_j induced by Ref on the j^{th} loop is (we use D_j as an intermediate variable):

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

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

All variables above represent integers.

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. Both imperfectly nested loops and non-unit loop step sizes are handled naturally without any special case. Hand-linearization of multidimensional arrays does not pose a problem, because the transformation 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. Only unknown loop bounds require additional transformation beyond that required in the basic framework [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 above for the minimum required the unroll factor U_j. We inherit the definitions in Section 4.2. Additional variables needed for 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 4.2) and the sum-of-products modulo theorem (Theorem 4.3). Then, 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 affine accesses is represented as a requirement of the step-size after unrolling is performed (Theorem 4.5). After that, 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 4.6). 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 4.7).

In all the proofs that follow, all variables and constants introduced are integers. The proof begins by supplying two supporting theorems involving modular arithmetic, theorems 4.2 and 4.3.

Theorem 4.2 requires Lemmas 1 and 2; stated below without proof.

Lemma 1 *Let X, N be integers, $X \geq 0, N \geq 1$. Let p_1, p_2 be integers that satisfy $N = \gcd(N, X)p_1$, $X = \gcd(N, X)p_2$. Then $p_1, p_2 \geq 1$ and are relatively prime.*

Lemma 2 *Let $X \geq 0, N \geq 1$, and let p_1, p_2 be those defined in Lemma 1. Then $\operatorname{lcm}(N, X) = p_1 X = p_2 N$.*

Theorem 4.2 (Product modulo theorem) *Given $X \geq 0$ and $N \geq 1$, let $S = \{s \mid sX \bmod N = 0\}$. Let D be the least element in S. Then $D = N/\gcd(N, X)$.*

Proof Let $p_1 = N/\gcd(N, X)$. We first show $p_1 \in S$: $p_1 X \bmod N = \operatorname{lcm}(N, X) \bmod N = 0$.

Next, we show that $\forall s \in S$, $p_1 \leq s$. $s \in S \Rightarrow sX = kN$ for some $k > 0 \Rightarrow sX \geq \operatorname{lcm}(N, X)$ $\Rightarrow sX \geq p_1 X \Rightarrow s \geq p_1$. □

Theorem 4.3 (Sum-of-products modulo theorem) *Given $n \geq 1$, $N \geq 1$, and $b_i \geq 0$ for $1 \leq i \leq n$. Then*

$$(k_1 b_1 + \cdots + k_n b_n) \bmod N = 0 \text{ for all } k_i \geq 0, 1 \leq i \leq n,$$

$$\Rightarrow b_i \bmod N = 0 \text{ for all } i, 1 \leq i \leq n.$$

Proof For each $i \in [1, n]$, set $k_i = 1$ and $k_j = 0, j \neq i$. Then $b_i \bmod N = 0$. □

Definition 4.4 *Given the representation of an affine access defined in Definition 4.1, let* $address(val_1, \ldots, val_k)$ *be the address of the affine function evaluated with index variables* $v_j = val_j$ $(1 \leq j \leq k)$*, assuming row-major array layout. Then,*

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

The following theorem derives the condition for memory bank disambiguation for an affine function access of the form in Definition 4.1.

Theorem 4.5 (Disambiguation condition) *Given the context in Definition 4.1, assume that the array of the affine access is low-order-interleaved. To get bank disambiguated accesses, each loop* $j \in [1, k]$ *can be unrolled so that the resultant step size* D_j *satisfies the equation:*

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

Proof Let X be an affine access in the loop nest before unrolling. Let X' be a static instance of X after unrolling. For that instance to be bank disambiguable, its address must obey the following:
$address(v_1, v_2, ...v_k) \bmod N = address(v'_1, v'_2, ..., v'_k) \bmod N$ for all legal input values
$\Rightarrow address(v_1, v_2, ...v_k) - address(v'_1, v'_2, ..., v'_k) = 0 \bmod N$.
Now, valid v_i, v'_i must have the form $l_i + m_iD_i$; $m_i \geq 0$. Substituting these values in and canceling common terms, we have
$address(m_1D_1, ..., m_kD_k) - address(0, ...0) = 0 \bmod N$; $m_1, ...m_k \geq 0$.
Thus,

$$\begin{aligned} &((\ldots((c_{1,1}MAX_2 + c_{2,1})MAX_3 + c_{3,1}) + \cdots + c_{d,1})\ m_1D_1\ + \\ &\ \vdots \\ &(\ldots((c_{1,k}MAX_2 + c_{2,k})MAX_3 + c_{3,k}) + \cdots + c_{d,k})\ m_kD_k) \bmod N = 0. \end{aligned}$$

By Theorem 4.3,

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

i.e.,

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

Using Theorem 4.2, 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 4.5 minimizes the overall code size for unrolling schemes that provide disambiguation.

Theorem 4.6 (Minimum code size) *The value of D_j in Theorem 4.5 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 $D_1 \ldots D_k$, *i.e.*, the product of the unrolled step sizes. From Theorem 4.5, the given value of D_j $(1 \le j \le k)$ 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 4.5. □

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 4.7 (Unroll factor formula) *In order to attain the value of D_j in Theorem 4.5, we need to unroll the* jth *loop nest* $(1 \le j \le k)$ *by a factor U_j given by* $U_j = \mathrm{lcm}(D_j, s_j) \ / \ s_j$.

Proof Unrolling a loop j produces step sizes that are multiples of s_j. From Theorem 4.5, an unrolled step size necessary for bank disambiguation is any multiple of D_j. Thus, the lowest attainable step size that results in disambiguation is $\mathrm{lcm}(D_j, s_j)$. The necessary unroll factor U_j to reach this step size is the unrolled step size divided by the initial step size: $U_j = \mathrm{lcm}(D_j, s_j)\ /\ s_j$. $\square$

4.4 Bounds in code growth 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. Note, however, that the unrolling required by modulo unrolling can often be combined with the unrolling used to expose instruction level parallelism. This combination can help reduce the unrolling overhead.

In this section, we first derive the worst case code growth. Then, we describe a padding optimization that can reduce the code growth. Finally, we present an example that 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 4.7 is provably at most N, the number of banks. From Theorem 4.5, $D_j = N/\textit{a positive integer} \leq N$. Inserting into Theorem 4.7, $U_j = \mathrm{lcm}(D_j, s_j)\ /s_j \leq D_j \cdot s_j/s_j = D_j \leq N$.

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 accesses, the overall code growth can often be limited to N irrespective of k by applying the padding optimization discussed later in this section.

A final observation regarding code growth is that the decision of whether to modulo unroll a nested loop is a local decision. If the code growth from modulo unrolling is deemed excessive for one nested loop, the compiler can choose not to unroll the loop without adversely effecting the modulo unrolling decisions in the rest of the program.

Padding Optimization For many affine functions that occur in practice, a simple optimization enables us to restrict the overall code growth and to 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 4.7, in the case when the padding optimization is performed.

Corollary 4.8 *In Theorem 4.7, 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 4.5, we get

$$\begin{aligned} D_j &= N \ / \ \gcd\ (N, ((\ldots((c_{1,j} MAX_2 + c_{2,j}) MAX_3 + c_{3,j}) + \cdots) MAX_d + c_{d,j})) \\ &= N \ / \ \gcd\ (N, \ (X \cdot MAX_d + c_{d,j})) \text{ for some integer X} \\ &= N \ / \ \gcd\ (N, c_{d,j}) \end{aligned}$$

It can be shown that since the value of X does not matter for this result, the result holds for cases when only last dimension of the array reference is affine. □

We define the following class of array references, which benefits from padding optimization.

Definition 4.9 (Simple-index last dimension) *A simple-index last dimension array reference is an array reference 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 4.10 *Consider a simple-index last dimension array reference. If the array accessed by the reference has its last dimension padded to a multiple of N, then at most one of enclosing loops needs to be unrolled. That is, the U_j values for the other loops will automatically be one.*

Proof From Corollary 4.8, $D_j = N \;/\; \gcd\,(N, c_{d,j})$. 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. □

Thus, for array references that have simple-index last dimension, Theorem 4.10 shows that the code-size growth is no more than N, irrespective of the depth of the loop nest.

In some cases, padding optimization may fail 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 predicted code growth is prohibitive, modulo unrolling can operate on a subset of array accesses to reduce the unroll factor – only the accesses in the subset will become bank disambiguated.

5 Results

This section presents results for the Maps memory system in the context of the Raw architecture. Application programs are compiled with Rawcc and simulated on a simulator of the Raw architecture as described in Section 2. The processing element on each Raw tile is a MIPS R4000 instruction set augmented with network access instructions. Latencies of the basic instructions are as follows: 2-cycle load, 1-cycle store, 1-cycle integer add or subtract; 2-cycle integer multiply; 36-cycle integer divide; 4-cycle floating add, subtract, or multiply; and 10-cycle floating divide. Except for divides, all basic floating point operations are fully pipelined. The simulator simulates infinite data and instruction memories on chip. Two sets of results are presented. Section 5.1 compares application performance with varying amount of bank disambiguation support. Results show that in the 32-tile case, Maps improves performance by a factor of 3 to 5 for a broad range of programs. Section 5.2 presents more detailed analysis of the performance with our bank disambiguation techniques.

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 1: Benchmark characteristics. Sequential time is the run-time for the single-tile case.

Application suite Table 1 lists the characteristics of the benchmarks used for the evaluation. The benchmarks are derived from the following sources: Rawbench [3], SPEC [9], Mediabench [15], UTDSP [16], Jade [24], UC Berkeley MPEG-1 Tools [26], and CHAOS [29]. 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. Rawcc compiles them without any user directives or pragmas of any kind. All speedups were attained with our automated compiler without user intervention. Because the Raw machine does not support double-precision floating point, all floating point operations are converted to single precision.

5.1 End-to-end performance

We first present results for end-to-end application performance with a varying degree of bank disambiguation support. In all cases, instruction level parallelism is extracted and exploited across the tiles of the Raw machine. Performance is collected for three types of disambiguation support: *trivial*, equivalence-class unification only (*ECU only*) , and *ECU with modulo unrolling*. In trivial support, the compiler has no intelligent disambiguation information. With no information, the compiler generally has two options: leave memory accesses undisambiguated, or perform trivial bank disambiguation by mapping all objects to one tile. On Raw, undisambiguated accesses happen to be very expensive due to software overhead [5], so we select trivial bank disambiguation as our baseline technique. This method models the cost of centralization in Raw's memory system, but it does not model the penalty due to extra capacity misses in a finite sized cache.

In *ECU only*, different equivalent classes provided by ECU are mapped to different tiles. In *ECU with modulo rolling*, non-array equivalent classes are distributed, and arrays accessed through affine references are low-order interleaved across the memory banks.

In these results, we expect end-to-end performance to improve with the quality of bank disambiguation for two reasons. First, bank disambiguation improves memory parallelism, allowing multiple concurrent memory accesses. Second, good disambiguation allows data to be "parallelized" with the computation, so that the data can reside close to the computation that uses it.

Figure 7 compares the performance of the three bank disambiguation schemes on a Raw machine with 32 tiles. The baseline for all three strategies is the execution time of the sequential program running on one tile, with one functional unit and one memory bank. The results show that ILP without memory parallelism yields poor performance. While using ILP alone gives a speedup in the range 1-4, memory parallelism can increase performance substantially. ECU increases the speedup on average by a factor of two beyond using ILP alone, boosting it to between 2 and 6.[3] Modulo unrolling further improves speedup to between 7 and 24 in applications where it is applicable. Overall, the methods in this paper deliver an additional factor of 3 to 5 in performance over using ILP alone for our benchmarks.

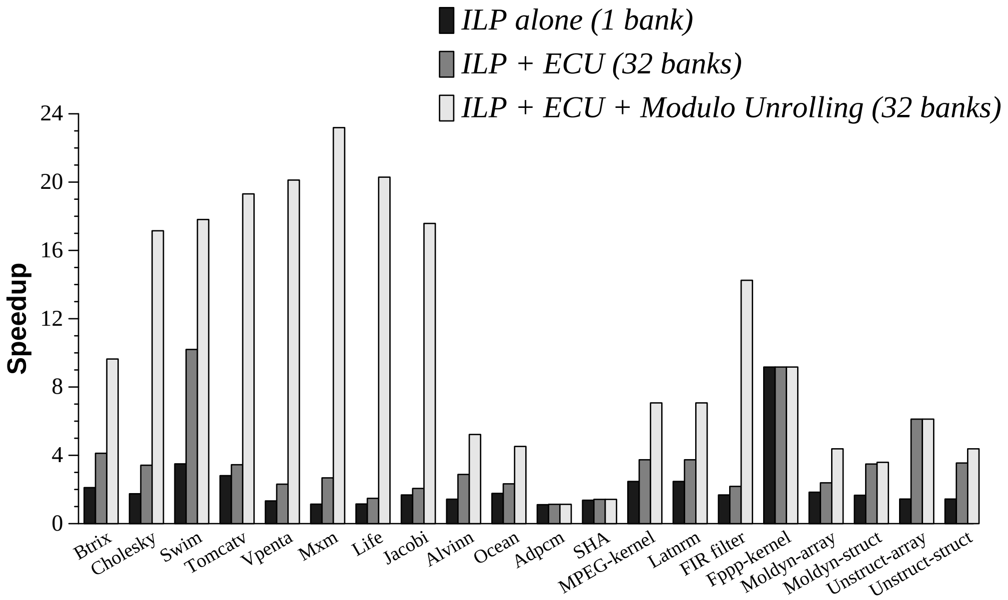

Figure 7: Comparison of 32-tile speedups using instruction-level parallelism (ILP) alone, ILP with ECU, and ILP with both ECU and modulo unrolling.

5.2 Detailed application results

Table 2 shows benchmark speedups for a varying number of tiles on Raw, using our bank disambiguation techniques. The numbers in the last column, for $N = 32$, are identical to the ILP + ECU + modulo unrolling numbers in Figure 7. We discuss some overall trends.

[3] Adpcm, SHA and fppp-kernel are exceptions; see Section 5.2 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 2: Benchmark speedup with full Maps bank disambiguation (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.

Benchmark performance can be classified into several types. Dense-matrix programs performed very well, attaining multiprocessor-like speedups on a microprocessor. The performance is due largely to modulo unrolling. For multimedia applications, two applications, Adpcm and SHA, attain low speedups while the remaining three attain high speedups. For Adpcm, the code is inherently serial for the large part; for SHA, while some ILP is available, it is too fine-grained for our current techniques to exploit. The other three multi-media applications benefit significantly from memory parallelism. The results on two of them, MPEG-kernel and FIR-filter, are especially encouraging as these are key components of the emerging workloads of the future involving audio, image and video data.

The irregular applications include Fppp-kernel, Moldyn and Unstructured. Fppp-kernel consists of a big time-intensive basic-block with much parallelism and mainly accesses to scalar data. Since only scalar variables are present, the benchmark does not benefit from ECU and modulo unrolling. Moldyn and Unstructured are more typical examples of scientific programs with irregular access patterns. Both moldyn and unstructured are run using two versions: one using structures, and the other using arrays instead for the structure fields. For these irregular applications, Maps is able to improve performance by a factor of 2 to 3 compared to using ILP alone.

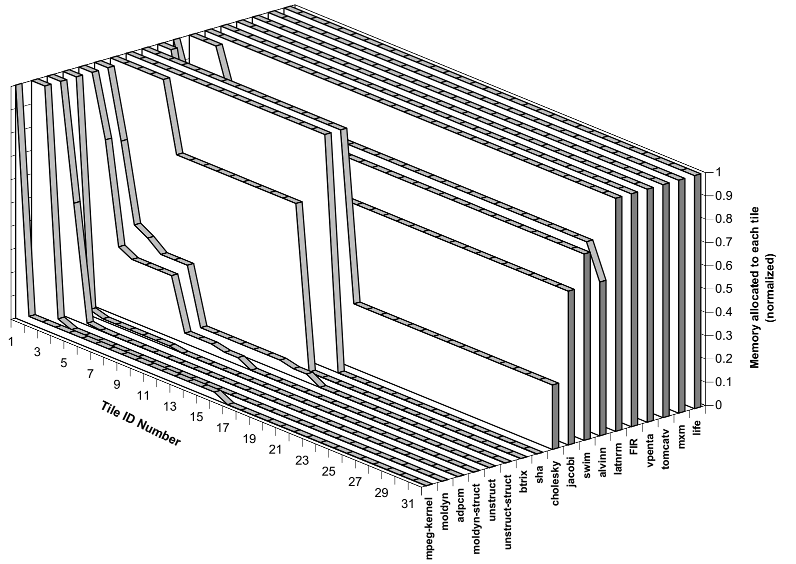

Figure 8: Distribution of primary data on a 32-tile Raw machine. The tiles are sorted in decreasing order of memory consumption. For each benchmark, the graph displays the memory consumption on each tile normalized by the memory consumption of the tile with the largest consumption.

Memory distribution We measure the distribution of memory across the tiles. 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 that is also fast. Figure 8 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. Load balance in the other applications is limited by two factors: the limited number of equivalence classes, and the unequal size of the classes.

Memory bandwidth utilization We measure how well an application takes advantage of Raw's independent memory banks. Bandwidth utilization depends on the amount of memory parallelism exposed by Maps and the amount of parallelism in the application. Figure 9 measures the weighted memory bandwidth utilization of a 32-tile machine. It plots

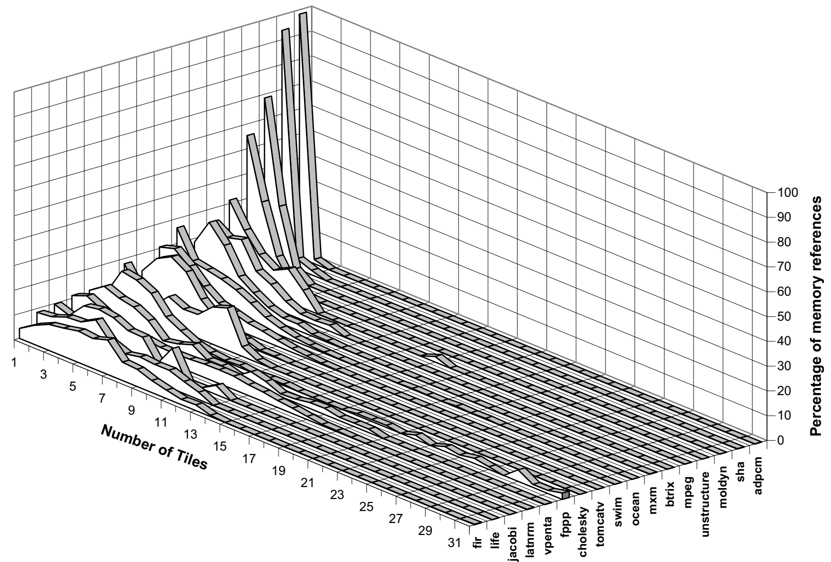

Figure 9: Weighted bandwidth utilization of the memory system on a 32-tile machine. The graph displays the percentage of memory references being issued in a time slot when a given number of tiles is issuing memory requests.

the percentage of memory references being issued in a clock cycle when a given number of tiles is simultaneously issuing memory requests. The sum of the percentages for any one application is 100%. For example, for Cholesky, almost 20% of the memory references are issued in a cycle in which a total of 7 memory references are 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. The figure shows that most of our benchmarks are indeed able to take advantage of the many ports in the memory system.

6 Related work

Bank-exposed general-purpose microprocessors date back to as early as 1983 when Josh Fisher proposed the ELI-512 VLIW machine [11]. The machine is a bank-exposed architecture with a point-to-point network connecting its processing elements, each with an exposed memory bank. It provides two ways to access memory: a fast "front door" that directly addresses a particular bank, and a slower "back door" that can address any bank. More recent bank-exposed architectures include iWarp [7] and the Raw machine [32]. On the embedded front, many DSP chips today, such as the Motorola DSP56000 family [20] and the NEC μPD7701x family [22], have two exposed memory banks, called X and Y memory.

Bank disambiguation is equivalent to memory bank prediction described by Fisher [11]. He explores memory bank prediction in the context of the ELI-512 VLIW machine. On this machine, successful bank disambiguation allows faster memory accesses through the front door. Fisher does not provide any automatic way to perform bank disambiguation, but he observes that unrolling can sometimes help disambiguate accesses. This observation forms our basic for modulo unrolling, our fully automated technique.

Since then, work on bank disambiguation has mostly been confined to the DSP community. Saghir, Chow, and Lee [28] have developed a method for exploiting digital-signal processing chips with dual memory bank. Their approach examines each memory reference and constructs an interference graph that represents how frequently data objects can be accessed in parallel. A min-cut algorithm is then used to partition the objects across the memories. Similarly, Sudarsanam and Malik [30] also exploit the use of dual memory bank in DSPs, with the additional constraint that each register is tied to a specific memory bank. They exploit both a greedy algorithm and a simulated annealing technique based on interference graph. Unlike Maps, these approaches do not deal with pointer aliasing, nor do they attempt to partition arrays. The common partitioning problem they solve are analogous to the problem of mapping virtual object partitions to physical tiles in the Raw compiler. In the Raw compiler, this problem is solved by the space-time scheduler [17], the back end of Maps.

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 [6, 13]. In addition, some irregular scientific applications can be parallelized on multiprocessors using inspector-executor method [10]. Typically these techniques involve user-inserted calls to a runtime library such as CHAOS and are not automatic [21]. 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. In contrast, the Maps approach exploits instruction level parallelism and is thus more generally applicable.

Literature includes many kinds of memory disambiguation. Most of them are unrelated to bank disambiguation, which is concerned with the location of a reference. Rather, they are usually concerned with the dependence relation between references. Disambiguation of this type includes relative memory disambiguation [18], run-time disambiguation [23], dynamic memory disambiguation [8, 12], and affine-memory disambiguation [2, 4, 19, 34].

7 Conclusion

This paper presents Maps, a memory system for bank-exposed architectures. Maps provides memory parallelism through a compiler-managed set of decentralized memory banks. This approach contrasts with the centralized view of memory maintained by existing microprocessors, which inhibits scalability due to the need for centralized dependence checking hardware and long wires. The system supports sequential programs and is transparent to the programmer, thus requiring no extra programming effort.

This paper focuses on compile-time bank disambiguation, the main problem in exploiting memory parallelism on a bank-exposed architecture. Bank disambiguation is the act of ensuring that a memory reference refers to data on only one memory bank. Two methods for banks disambiguation are presented. Equivalence class unification uses pointer analysis to partition data into classes that can be mapped to different banks without disturbing bank disambiguation. Modulo unrolling uses unrolling to enable the bank disambiguation of affine accesses to low-order interleaved arrays.

We are encouraged by the results of the Maps approach to providing memory parallelism. Experimental results demonstrate that our disambiguation methods improve performance by a factor of three to five. 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 can benefit from our techniques.

References

[1] V. Agarwal, M. S. Hrishikesh, S. Keckler, and D. Burger. Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures. In *Proceedings of the 27rd International Symposium on Computer Architecture (ISCA'00)*, pages 248–259, June 2000.

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

[3] J. Babb, M. Frank, V. Lee, E. Waingold, R. Barua, M. Taylor, J. Kim, S. Devabhaktuni, and A. 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.

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

[5] R. Barua. *Maps: A Compiler-Managed Memory System for Software-Exposed Architectures.* PhD thesis, M.I.T., Department of Electrical Engineering and Computer Science, January 2000.

[6] W. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, B. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford. Effective Automatic Parallelization with Polaris. *International Journal of Parallel Programming*, May 1995.

[7] 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.

[8] W. 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.

[9] S. P. E. Corporation. The SPEC benchmark suites. *http://www.spec.org/.*

[10] R. Das, M. Uysal, J. Saltz, and Y.-S. Hwang. Communication Optimizations for Irregular Scientific Computations on Distributed Memory Architectures. *Journal of Parallel and Distributed Computing*, 22(3), September 1994.

[11] J. 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.

[12] D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, and W. 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.

[13] 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, Dec. 1996.

[14] M. Horowitz, R. Ho, and K. Mai. The Future of Wires. *Semiconductor Research Corporation Workshop on Interconnects for Systems on a Chip*, May 1999.

[15] 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.

[16] C. Lee and M. Stoodley. UTDSP BenchMark Suite. *http://www.eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.html*, 1992.

[17] W. Lee, R. Barua, M. Frank, D. Srikrishna, J. Babb, V. Sarkar, and S. 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, Oct. 1998.

[18] 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, Jan. 1993.

[19] D. 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.

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

[21] S. Mukherjee, S. Sharma, M. Hill, J. Larus, A. Rogers, and J. 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.

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

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

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

[25] International Technology Roadmap for Semiconductors, 1999 Edition. *Semiconductor Industry Association*, 1999.

[26] L. A. Rowe, K. Gong, E. Hung, K. Patel, S. Smoot, and D. Wallach. Berkeley MPEG Tools. *http://bmrc.berkeley.edu/frame/research/mpeg/*.

[27] 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.

[28] M. A. R. Saghir, P. Chow, and C. 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.

[29] J. Saltz, R. Ponnusamy, S. Sharma, B. Moon, Y.-S. Hwang, M. Uysal, and R. Das. A Manual for the CHAOS Runtime Library. Technical report, University of Maryland: Department of Computer Science and UMIACS, March 1995.

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

[31] D. Sylvester and K. Keutzer. Rethinking Deep-Submicron Circuit Design. *IEEE Computer*, 22(11):25–33, November 1999.

[32] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring It All to Software: Raw Machines. *IEEE Computer*, 30(9):86–93, Sept. 1997. Also available as MIT-LCS-TR-709.

[33] R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S.-W. Liao, C.-W. Tseng, M. Hall, M. Lam, and J. Hennessy. SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers. *ACM SIGPLAN Notices*, 29(12), Dec. 1996.

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