Hot Pages: Software Caching for Raw Microprocessors

Csaba Andras Moritz   Matthew Frank
Walter Lee   Saman Amarasinghe
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139
{andras,mfrank,walt,saman}@lcs.mit.edu

Abstract

This paper describes *Hot Pages*, a software solution for managing on-chip data on the Raw Machine, a scalable, parallel, microprocessor architecture. This software system transparently manages the mapping between the program address space and on-chip memory. Hot Pages combines compile time information to selectively virtualize memory references and to eliminate many cache-tag lookups. For many of the memory accesses that cannot be fully predicted, Hot Pages replaces the cache-tag lookups with simple register comparisons by reusing translated virtual page descriptions from earlier nearby memory references. Hot Pages implements a multi-bank memory structure, allowing multiple references in parallel, to provide memory bandwidth matched to the computational resources on the Raw microprocessor. Because virtualization is handled in software rather than hardware, the system is easier to test, it is more predictable, and provides the flexibility of application specific customized caching solutions. For the applications studied the Hot Pages system eliminates in average more than 90% of the cache-tag lookups and could be applied to reduce the power required for data caching. The performance of Hot Pages scales with added processors and for many applications is comparable with that of hardware solutions. Hot Pages is a credible new foundation for caching, opening up a new dimension for research in additional application specific software caching optimizations.

1 Introduction

Rapid advancement in semiconductor technology allows the integration of more functional units and large memories on a single-chip. The increased hardware complexity of these systems makes the design and verification process challenging and costly. In addition, as on-chip devices shrink, on-chip wires are becoming slower relative to logic so the die area that is reachable in one clock-cycle is decreasing. Architectures that require long wires will not be able to scale up with technology. In this paper we propose a system that completely eliminates the caching hardware using instead a simple software-only solution.

The most radical trend change in microprocessor design is perhaps due to the emergence of small hand-held systems connected to the Internet. These systems favor simple designs with low power consumption, low cost and fast time to market.

The processing workload of these systems is also changing, from the traditional non-numeric desktop applications to streaming applications such as image compression, 3D graphics, speech recognition, and broadband communication. Streaming applications have typically poor cache behavior and cannot fully benefit from a hardware caching implementation. An adaptive, application specific software caching solution would be a feasible alternative to hardware caches. A software solution not only could adapt to application requirements but also to performance requirements, thus better price/performance ratio could be achieved.

Memory management and caching accounts for a good fraction of power consumption in a microprocessor, *e.g.* in a low power StrongARM SA110 microprocessor 8% of the power is consumed by the TLB management and 18% by the data cache [2]. In table 1 we show the estimated area and power consumption of a direct mapped and a two-way set associative hardware cache using the same IBM ASIC technology as used for the Raw prototype. A hybrid software-hardware memory management solution could exploit the static information available during compilation to turn off the tag memory and portions of the cache to optimize the power required for memory management. A first estimation shows that a software exposed hardware cache could potentially save half of the power required for data caching.

Motivated by these trends we developed a system called Hot Pages, a software-only solution for managing on-chip data caching on the Raw Machine, a scalable, software exposed, microprocessor architecture.

The challenge of a software based approach to caching is to balance the tradeoffs between the added software overheads against opportunities provided by a caching scheme specialized for each application. Our technique, called *Hot Pages*, combines both compile-time and runtime techniques to provide completely transparent virtualization. It reduces the software overheads by mapping parts of memory accesses into non-cached SRAM memory, by eliminating many of the cache-tag lookups and by optimizing the hit case by caching (saving) translated virtual page descriptions likely to be reused for nearby memory references. In our terminology, a virtual page is a contiguous address range in DRAM that is mapped into the SRAM of one tile. A physical page in the SRAM is similar to a cache-line or block in hardware caches.

The specific contribution of this paper include:

- Design and evaluation of a fully automated software-only solution for caching.
- Development of novel compile-time techniques used to eliminate tag checks and reduce/mask the overhead of software schemes.
- A new foundation for caching, a software-only cache safety-net that enables many other cache optimizations.

Our results demonstrate that compile time analysis can eliminate a large portion (90% in average for the applications studied) of the cache-tag lookups. While having a small/fast memory near the processor gives several orders of magnitude performance improvement, the extra hardware that performs cache-tag checks

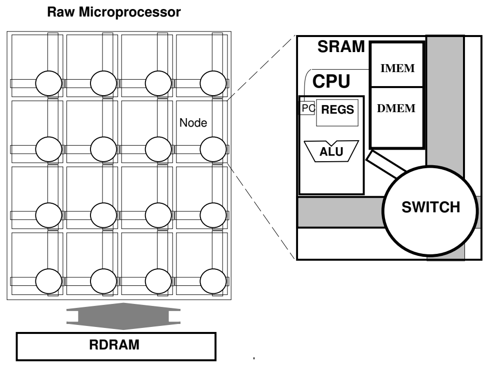

Figure 1: Raw system composition. A typical Raw system includes a Raw microprocessor coupled with off-chip DRAM and stream IO devices. Each Raw tile contains a simple RISC-like processor, an SRAM memory for instructions and data, and a switch. The tiles are interconnected in a 2D mesh network that is space-time orchestrated by the compiler.

Cache	Cache memory area	Tag area
Direct Mapped	510063	77944
Two-way Set Associative	510063 (692pf)	85589 (247pf)

Table 1: Cost and power consumption of a direct mapped and a two-way set associative hardware cache including all circuitry using similar ASIC technology as the Raw chip. The configuration assumed is 32byte cache line and 32 kbyte cache. The direct mapped cache is using 8kx32 SRAM for the cache and 1kx27 SRAM for the tag memory. The two-way associative cache is using one 4kx64 SRAM and a 512x27x2 SRAM for the tag memory. We indicate the areas in standard IBM cell counts, power is shown in pf. Based on these numbers we could speculate that a software controlled hardware cache with 70% of the tag-checks eliminated and using 2 4kx32 SRAM for the 2-way set associative case each with 407pf energy consumption would consume 407pf + 0.3x274pf= 489pf, equivalent to a 50% saving in power. Also note, that the tag area is 15%-17% of the area required to implement the cache corresponding to 5% of the chip area. Although this area cannot be saved for all the software caching configurations (because the software mapping requires memory too), it can potentially be saved (and used for example for the local stack accesses) for applications where a larger cache-line size can be used.

gives a relatively small additional improvement. The technique of using compile-time information to reduce the number of tag-checks could be used to reduce power in low-power systems or to improve hit-ratio in hardware caches. Assuming an 18% energy consumption for data-caching similar to a StrongARM system, a 10% total power reduction could be achieved.

For the applications studied the performance of Hot Pages scales with added processors and for many applications is comparable with that of hardware solutions. There are two categories of applications where software caching obtains competitive performance. First, there are fully analyzable applications where the Hot Pages predictable scheme succeeds in eliminating most of the cache-tag lookups while not requiring additional runtime overhead. Second, is the case of applications with lower cache-hit ratios and many conflict misses. This is especially true when compared to a simple direct mapped cache (often the preferred choice as a first level cache) with fixed cache-line size. The software caching solution can easily implement more associativity, select an appropriate cache-line size based on application memory requirements and reduce the number of virtualized memory references.

Note, that a larger cache-line size that is not causing conflict misses is in general advantageous because it reduces the number of times the chip is traversed and the DRAM is accessed improving overall performance [13]. Choosing the optimal cache-line size is going to be more significant in future generations of chips with larger number of tiles because of the increased average latency required to traverse the chip. We can expect the widening speed gap between processors and DRAMs to further penalize DRAM accesses. Thus, as in current virtual memory systems, the cost of sophisticated software-based techniques can be easily amortized by the gain in the improved hit rates of the next-generation caches.

We believe that these trends and additional compiler techniques will further increase the range of applications where software customization of caching strategies can provide enough benefits to compensate the extra overhead introduced. While the Hot Pages system described in this paper is evaluated and targeted for Raw, its core techniques can be adapted to other microprocessor and multiprocessor systems.

1.1 The Raw Machine

A typical Raw system (see figure 1) might include a Raw microprocessor [18, 13], coupled with off-chip RDRAM (RamBus DRAM) through multiple high bandwidth IO paths. A Raw microprocessor is based on a simple mesh-connected set of tiles. Each Raw tile contains a simple RISC-like processing core and an SRAM memory for instructions and data. The two level memory hierarchy, namely, a local SRAM memory attached to each tile inside the Raw chip, and a large external RDRAM memory, is necessary to solve large problems that exceed the size of the on-chip memory. SRAM memory distributed across the tiles eliminates the memory bandwidth bottleneck, provides low latency to each memory module, and prevents off-chip I/O latency from limiting effective computational throughput.

The Raw microprocessor achieves good performance by using a compiler to find instruction level and memory parallelism. The Raw architecture requires only short wires, provides both memory and computational parallelism to support streaming applications, and because its hardware structure is simple, it is easier to design than superscalar microprocessors.

The remainder of this paper is organized as follows. Section 2 describes the design goals and components of the software caching system. Section 3 presents the compiler steps and optimizations implemented for the Hot Pages schemes. Section 4 gives our experimental results. Section 5 contains the related work. Section 6 concludes the paper.

2 Software Caching

This section describes the design goals and the main components of the Hot Pages system. The design goals include: (1) scalability or efficient use of compile-time information to provide simultaneous SRAM and DRAM accesses, (2) support for the different types of memory accesses such as static, dynamic, speculative, and (3) leveraging static information to reduce/mask the overhead of the software solution. The latter is an important goal also because it is a prerequisite in supporting more sophisticated specialized resource management.

Address translation is the process of mapping a program address on a Raw tile to an SRAM address. If the compiler can predict the physical page the memory reference is accessing then it can also generate the correct SRAM address without requiring further runtime check. We call this optimization Hot Pages *predictable*. Another novel aspect of our system is the compile-time analysis called Hot Pages *likely* that identifies if a program memory reference can reuse a previous virtual page translation. This is a very important optimization because it can be applied to reduce the cost of the hit case of accesses that cannot be fully predicted at compile time. We call such a virtual page a *hotpage* for two reasons: (1) the compiler guarantees that its translation is saved into registers for fast access (eliminating the need of tag-checks if prediction is accurate), and (2) the compiler identifies other memory accesses that likely can reuse that translation. The address translation for a hotpage reference is customized at compile-time. It uses directly the specific hotpage page translation (saved in registers). References mapped to hotpages with the likely optimization require no table lookup or hashing during runtime if predicted correctly. Several hotpage descriptions (translations) can coexist at the same time. Besides lowering software overheads, this solution provides opportunities for further optimizations as it can potentially mask the overhead of more sophisticated memory management schemes, i.e. inverted page table organizations and replacement policies that require updating during the common hit-case.

2.1 Runtime Mechanisms

The runtime system can be divided into four components, (1) address translation, (2) page table organizations, (3)off-chip communication and (4) replacement mechanism.

Address Translation Address translation is the mapping of program addresses into SRAM addresses. This mapping can be implemented by using a page table lookup similar to virtual memory systems. The mapping is done at a *page* granularity. A page is a contiguous address range in both SRAM and global virtual memory. Address translation implemented in software adds significant amount of software overhead to the execution time. The translation overhead varies for different page table organizations. When the program performs a load or store to a virtual address, a piece of software translates this into an SRAM address using the mapping information in the page table. In section 3 we describe situations where the address translation can be simplified. If the translation results in an access on a page that is not currently in SRAM, a software *page-fault* handler is invoked. The handler implements a dynamic messaging protocol to fetch the page from the off-chip DRAM. Given, that the DRAM is divided into several banks and the virtual address spaces seen by individual tiles are distinct, simultaneous DRAM accesses can be supported.

Software Managed Off-chip Accesses

In the system presented in this paper we use the Raw tiles along the edge of the chip (which we also call IO tiles) to set up the off-chip accesses. The final Raw prototype will also have a path to access the external pins directly from the interior tiles, but we have not yet converted our system to use this faster path to external memory. An execution tile (or the tile executing the miss handler) sends a request through the network to the IO tile that in turn accesses the DRAM. The communication is fully pipelined between the execution tile, IO tile, and DRAM, requiring very little processing overhead and no intermediate buffering. Designing a communication solution without extra buffering is a prerequisite to be able to have large cache-line sizes. For example, the total cost of the miss handler with 256 words cache line fetch operation is 590 cycles equivalent with a throughput of 2.3 cycles per word for off-chip reads. The breakdown of costs for several cache-line sizes is presented in table 2.

Cache-line	Fetch	Write-back and fetch
512w	1252	1935
256w	590	1124
128w	444	724
32w	193	405

Table 2: Miss-handler costs in cycles for various cache-line sizes in a software direct mapped cache. Note, that these costs includes the cost of fragmenting messages into 16 word packets (due to the size of network buffers that only allow 16words message payload), as well as the software overhead of IO tiles. The DRAM latency is 3 cycles, crossing the chip boundary takes 2 cycles, average latency inside the chip is 4 cycles. There is a 14 cycle procedure call overhead and a 19 cycle overhead associated with housekeeping operations, updating the mapping tables, selecting the DRAM bank for the request and initial setup of message headers.

Page Table Organizations The mapping of virtual addresses into physical SRAM addresses is organized into page tables, which are collections of *page table entries* (PTEs). In our system, a PTE has a minimal function indicating only whether its virtual page is in SRAM or not. Modern virtual memory systems use page tables to handle additional functions such as protection and sharing that our software caching system does not need to implement. Figures 2 and 3 show two examples of page table organizations we implemented. A software implementation of a page table implies new tradeoffs. For example, the single-level fully mapped page table would require less software overhead but a larger amount of memory for address translation purposes than an inverted page table with set-associative mapping. Note, that an inverted page table organization has a page table size proportional with the physical SRAM size having fewer PTEs. Although more associativity in a page table scheme can reduce conflict misses, such an organization can potentially add more overhead to the common case hit-path (because of increased number of PTE lookups). Our direct mapped caching scheme uses an inverted page table with PTEs indexed with the physical frame number. The PTEs in the direct mapped scheme correspond to virtual page numbers. The larger memory required for fully mapped tables makes the fully mapped solution practical only when used with large page sizes and applications with small memory requirement.

For example, a 32 Kbyte SRAM with 256 word cache line size would require only 32 words for the inverted page table mapping, independent of the application memory requirement. A similar fully mapped table will have a size dependent on the number of tiles and the application. Assuming a 16 tile configuration and an

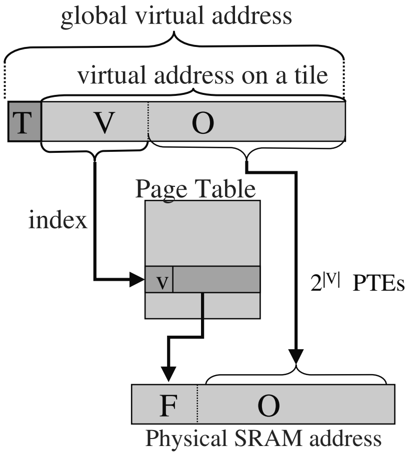

Figure 2: Address translation using a fully-mapped single-level page table. The main advantage of this organization is its relatively low overhead for a page table entry lookup. The disadvantage is the memory space required for the entries which is reflecting the size of the virtual address space. Notation: *T* = TileID, *V*= Virtual Page Number, *O*=Offset in Page, *v*=valid bit, *F*= Physical SRAM Frame.

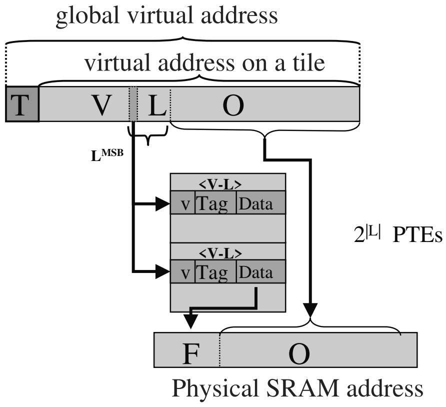

Figure 3: Address translation using an inverted page table with two-way set associative mapping. This organization has low memory requirement for page table entries, proportional with physical SRAM memory , but has a higher translation overhead also because of the necessary tag checking and the potential lookup of multiple entries. Notation: *T* = TileID, *V*= Virtual page number, *L*= low order bits of V corresponding to physical page numbers in a direct mapped SRAM, *O*=Offset in page, *v*=valid bit, *F*= physical page Frame.

application with 2 Mbyte data footprint such a mapping requires a 128 words page table.

We have also implemented dynamic column-associative software caching schemes inspired from the column-associative schemes presented in [1]. The main objective of the column associative hardware caches is to improve the hit ratio of direct mapped caches while keeping the latency of the primary hit case equal to that of a direct mapped cache. A low software overhead corresponding to a direct mapped cache lookup but a better replacement are desirable features for software mapping schemes. However, the hardware schemes mentioned earlier assume that swapping of entire cache lines can be done efficiently. In a software solution swapping of larger cache lines is not practical. Our solution uses indirect mapping and swapping of table entries instead of entire cache-lines. Our basic scheme that we call *hash-rehash software cache* has a software overhead for the hit case similar to that of the direct mapped cache but with improved hit rate. On a primary miss our scheme selects a different hash function- rehashing. In contrast to the hardware scheme we do not use a hash function based on bit-selection for the primary lookup because such a scheme incurs additional 2 cycles software overhead for the common hit case. We use the rehashing bit-flipping function, i.e. inverting the highest order bit of the index to select the secondary PTE. In order to distinguish between PTEs that correspond to rehashed indexes we use two words per entry in the page table instead of one (alternatively we could have encoded in one word but would have needed to extract the bits instead). The first word in the PTE corresponds to the virtual page number and the second word to the index before the rehashing function was applied. This is necessary in order to be able to swap the primary and secondary entries during replacement. Swapping on a secondary miss occasionally is required because otherwise the primary entries would never be replaced.

With additional overhead during the hit-time it is possible to implement a scheme similar to the hardware rehash-bit scheme to eliminate secondary hashing [1] and further improve the hit rate.

Replacement Mechanism Our system currently implements a circular FIFO replacement policy for the fully-mapped case. This policy has the advantage of not adding overhead to the common case hit-path as it only requires update of the replacement data-structure during the miss-case. Page entries that correspond to hotpages are non-replaceable and are ignored during replacement. In the direct mapped cache there is only one page that can be replaced. If we map hotpages through this table we have no other option but to replace them in case of a conflict situation. However, we can rely on pointer analysis to guarantee that non hotpage accesses will not access pages that currently map to hotpages and therefore we can move the mapping of hotpages outside the main mapping table. In the dynamic column associative scheme we currently choose to have a random replacement.

3 Compiler

Rawcc, the base Raw compiler, takes a sequential C or Fortran program and generates parallelized code which can be executed on the Raw machine. Parallelization involves the distribution of instructions and data across tiles and the management of communication and synchronization between the tiles. For details of this system, please refer to [4] and [11].

The base Raw compiler generates code assuming infinite memory per tile. Our Hot Pages system removes this assumption by implementing a virtual memory abstraction in software. Figure 5 shows the flow of the augmented Raw compiler which includes the new phases required for supporting virtualization and software caching.

This section describes the compiler components of the Hot Pages system in detail. Section 3.1 describes pointer analysis, an analysis technique used to determine the location set list of each memory reference. Section 3.2 describes how the Raw compiler distributes data across the chip and guarantees that the virtual addresses on individual tiles are never overlapped. Section 3.3 describes the hotpage analysis phase which divides memory references into groups called *hotpage sets*. All references inside a hotpage set will essentially use the translation saved for a specific virtual page called a *hotpage*. All the memory accesses that are cacheable are changed with library procedure calls in the virtualization pass. A short description of this pass is presented in section 3.4. Note, that both the analysis phases and the virtualization

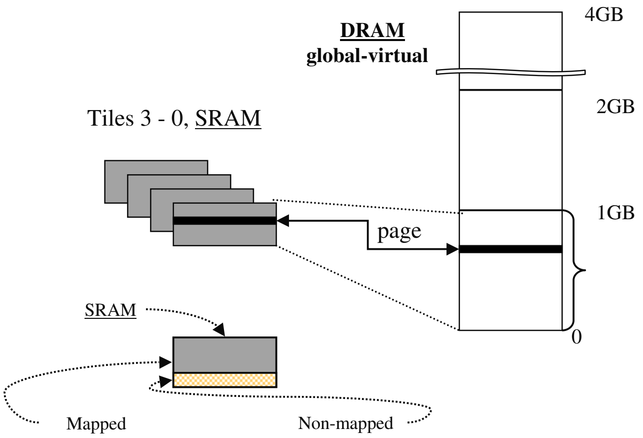

Figure 4: Global and Tile Virtual Memory. Note, that the memory disambiguation and data distribution phases in the compiler are guaranteeing that the virtual spaces on different tiles are non-overlapped. For example, each tile in the figure has 1GB of virtual address space. Tile0's virtual address space is mapped to the first GB, Tile1's virtual address space corresponds to the next 1GB address range and so on. The SRAM on each tile is divided into a mapped (cacheable) and a non-mapped (non-cacheable) region. The non-mapped area is further divided into a local stack area and a system area containing the mapping table and related data structures. The non-mapped memory is used to provide fast access (1 cycle) for the majority of the local stack accesses.

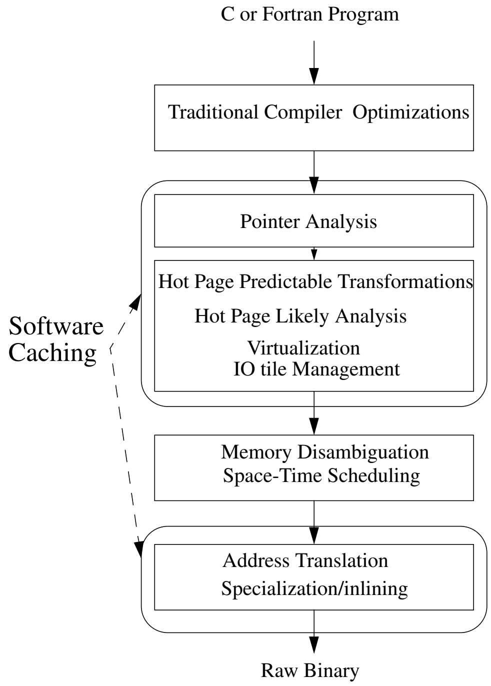

Figure 5: Structure of the Raw Compiler with Software Caching

pass are performed at a very early stage in the compiler. The specialization step happens in the compiler backend when both program and data is already partitioned and distributed accross the tiles. Clearly, without the propagated annotations from the frontend analysis passes we would not be able to specialize the translation for the individual program references. The specialization happens in the same time with inlining of an optimized version of the code required for the hit-case. The inlining is necessary to eliminate the call overhead for the code that is in the critical-path during memory accesses. Specialization (section 3.5)is an important pass as it eliminates the need for runtime checks in case of multiple hotpage sets. This compiler controlled approach is different from a software TLB solution where TLB entries would need to be looked up at runtime. Figure 6 introduces an example which illustrates the steps the compiler performs for software caching. The Hot Pages predictable compiler technique used to fully eliminate the runtime check for some of the memory references is presented in section 3.6.

3.1 Pointer Analysis

Pointer analysis is a compiler analysis which finds a conservative estimation for the set of data objects that a memory reference can refer to. The analysis is conservative in that some objects in the set may not be referenced. One standard application of pointer analysis is to determine dependence between memory references. In our software caching system, the analysis is used to guide the placement of data and for the hot page optimization.

Pointer analysis is provided by SPAN, a state-of-the-art pointer analysis package [14]. It disambiguates between accesses to abstract objects. An abstract object is either a static program object, or it is a group of dynamic objects created by the same memory allocation call in the static program. An entire array is considered a single object, but each field in a struct is considered a separate object. Pointer analysis identifies each abstract object by a unique *location set number*. It then annotates each memory reference with a *location set list*, a list of location set numbers corresponding to the objects that the memory reference can refer to. Object creation sites are annotated with their assigned location set numbers. A struct is marked with multiple location set numbers, one for each of its field. For simplicity, location set numbers are assigned only to non-pointer objects; in reality all objects are assigned such numbers. Each memory reference is marked with the location set numbers of the objects it can reference.

3.2 Data partitioning and Memory Disambiguation

The primary objective of Raw's data partitioning approach is to make the memory access pattern static, or compiler predictable. A *static memory reference* is a memory reference which refers to the memory on only one tile. A static load or store is performed by placing the load or store directly in the instruction stream of tile on which its memory resides. This type of references offers two advantages. First, it enables the Raw compiler to orchestate the communication between the memory reference and its depend instructions. Such orchestration is an order of magnitude faster than the mechanism for a non-static access. Second, static memory accesses simplify the process of memory virtualization. To check whether data for a static access is available on-chip, one only needs to consult the local page table on its resident tile rather than a global one. The global virtual address space is the sum of the tile virtual address spaces. Figure 4 shows a *4GB* virtual memory space (or DRAM) that is partitioned equally between the four tiles. A portion of each of the tile's SRAM is non-mapped, and is used for memory accesses that are not cached. This allows the compiler to optimize the memory access cost (because address translation is not required) for local stack references.

The process of creating static accesses through intelligent data partitioning and code transformation is called *static promotion*.

```
int A[10000];
int B[10000];


void main( ) {
int *pB;
int j,temp;
pB = &B[0];
for (j = 0; j < 10000; j++) {
     temp = *pB++;
    A[j] = temp;
 }
}
```

Pointer-analysis

A.)

```
int A[10000]; //hp2
int B[10000]; //hp1


void main( ) {
int *pB;
int j,temp;
pB = &B[0];
for (j = 0; j < 10000; j++) {
     temp = *pB++; //hp1
    A[j] = temp;      //hp2
 }
}
```

B.)

```
int A[10000]; //hp2
int B[10000]; //hp1


void main( ) {
int *pB;
int j,temp;
pB = &B[0];
for (j = 0; j < 10000; j++) {
     temp = CheckLoadVM(pB++); //hp1
    CheckStoreVM(&A[j], temp);   //hp2
 }
}
```

C.)

```
Tile 0                                   Tile 1
                                         …….
move reg4, vaddr0_1                      move reg4, vaddr1_1
jal CheckLoadVM        //hp1             jal CheckLoadVM       //hp1
move reg4, vaddr0_2                      move reg4, vaddr1_2
move reg5, reg2                          move reg5, reg2
jal CheckStoreVM       //hp2             jal CheckStoreVM      //hp2
…….                                      …...
```

D.)

Figure 6: An example of how Hot Pages implements software caching in the compiler: A.) Pointer analysis is used to determine the location sets of memory references, for example *pB* has the same set as *B*, B.) the *Hot Pages likely* analysis annotates memory references into hotpage sets *hp1, hp2*, C.) the *virtualization* pass changes the memory references with procedure calls, and D.) the address translation *specialization* pass in the compiler back-end inlines *specialized* code for *CheckLoadVM, CheckStoreVM*. This specialization is controlled by the hotpage set annotations (i.e., *hp1, hp2*). Note that the hotpage sets on Tile0 and Tile1 are different.

Maps, the memory management component of Rawcc uses two techniques to perform static promotion: equivalence class unification and modulo unrolling. Equivalence class unification (ECU) uses pointer analysis to help partition data objects such that every reference only refers to objects in one partition. Each of these partitions can then be placed on a single tile. Modulo unrolling allows arrays, which pointer analysis treats as a single object, to be both distributed and accessed statically. First, arrays are distributed through low-order interleaving. In this scheme, consecutive elements of an array are interleaved in a round-robin manner across the memories of the Raw tiles. Then, all affine accesses to those arrays can be transformed into static accesses through unrolling. For a more detailed description of these techniques, please refer to [4] and [3].

3.3 Hot Pages Likely Analysis

In this section we describe the compiler analysis for Hot Pages likely. Our analysis is performed on the Stanford University Intermediate Format (SUIF). THe Hot Pages likely analysis leverages the information provided by the pointer analysis pass to determine the location set of all memory accesses. The objective of the analysis is to identify if a program memory reference can reuse a previously translated virtual page description. Address translation in software is a very costly process. Even in the case of the simplest translation procedure we need to index into a table, read the entry, determine the validity of the entry, extract the data required for constructing the page frame, and finally combine the physical frame together with the offset. Without compiler analysis we would need to do full address translation for each of the memory accesses in the program. This overhead unfortunately is added to the common case hit path. The use of replacement policies such as Least Recently Used (LRU) and Least Frequently Used (LFU) exacerbate this problem as they require updating of a data-structure during the hit-case.

Our technique leverages static information about the locality of accesses to be able to implement a faster address translation. The compiler groups memory accesses into groups called *hotpage sets*. Hotpage sets are determined based on their location sets (information given by pointer analysis) in the memory. Each hotpage set contains references that can likely reuse the address translation saved for a specific virtual page called *hotpage*. The compiler algorithm has two phases. First, it finds data objects such as arrays and structures and map these objects to hotpage sets. Then, it traverses the control-flow graph of the program and maps memory accesses to existing hotpage sets based on their location sets. For example, if the compiler determines from the location set information that a load is accessing a location from a memory area allocated to an array, then it can likely reuse the address translation saved for the hotpage set assigned to that array. Note, that several virtual pages can be *hot* at the same time. We call this analysis *Hot Pages likely*.

There are some accesses (i.e. affine array accesses, scalars, stack frames) where the compiler can fully guarantee that the access goes to a certain page. This can be achieved by controlling the data mapping and by performing specific program transformations similar to strip-mining. We call this analysis *Hot Pages predictable*. The difference between the two translations is shown in Figure 7. If an access is annotated with the likely annotation then the actual virtual page number needs to be compared with the virtual page saved for the hotpage. This runtime check is not necessary for the predictable case.

Each hotpage has two pieces of information assigned to it: a

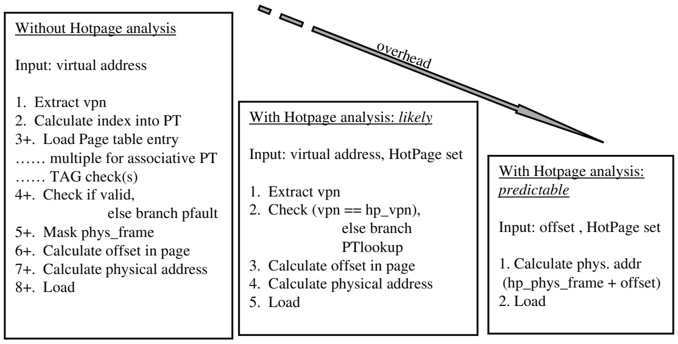

Figure 7: Address translation: (1) Without compile-time support, (2) With Hot Pages likely, and (3) With Hot Pages predictable. Notation: *vpn*= Virtual Page Number, hp_vpn= register containing the Virtual Page Number for a hotpage set, hp_phys_frame= register containing the Physical Page Frame for a hotpage set. Several hotpage sets can coexist at the same time each using separate hp_phys_frame and hp_vpn registers.

virtual page number and a physical frame, both saved into registers. A hotpage set miss-prediction will add two instruction overhead to the normal (or full) address translation, namely, the virtual page number and the new translated physical frame are updated for that particular hotpage set. To reduce register pressure our compiler currently uses three (number determined empirically) hotpage sets, and reuses these for different location sets in different regions in the program.

3.4 Virtualization

The compiler determines which accesses should be cached in the virtualization phase. If an access is virtualized then it is substituted with a procedure call into the virtual memory system. The compiler can decide not to virtualize an access and map it to a non-cached (non-mapped) SRAM memory area. These accesses are local and only cost 1 cycle. Typical accesses that are chosen not to be virtualized in the compiler are for example local stack references. Similarly, register spills are also mapped to this non-cached memory. Larger local objects are however virtualized to reduce the memory required for the non-cached memory.

3.5 Specialization

The address translation for hotpage references is customized at compile-time directly for the specific hotpage page description (translation). No extra table lookup or hashing for accessing a hotpage description is required at runtime. The address translation specialization is done in the compiler-backend on the Machsuif intermediate format. At that stage in compilation, the sequential program has already been parallelized and the data has been distributed. The compiler virtualizes the accesses that have to be translated at runtime, and annotates each access with its type and with the hotpage set the access belongs to. Based on these annotations an optimized version of the hit case handler is inserted. These handlers are using the registers that contain the translation for the specific hotpage sets. The miss-handlers are kept as runtime library calls. The hit-case handlers are implemented using virtual registers, therefore no register life range analysis is needed. This approach reduces register pressure because the inlined address translation code (hit-case) is register allocated together with the rest of the program.

3.6 Hot Pages Predictable Optimizations

The Hot Pages likely system reduces the tag-check lookup to only a few operations. The main objective behind the Hot Pages predictable optimization is to completely eliminate the virtualization software overhead for certain classes of memory accesses analyzable in the compiler. There are several possible approaches to this (see for example [8]). Our current system targets inner loops with affine array accesses.

The idea is to strip-mine each inner loop of the program into double nested loops, in such a way that we can guarantee that the memory references from the new innermost loop are only accessing memory locations *known to be in the cache*. We currently determine the *safe* upper bound of the innermost loop at runtime. A copy of the body of the original inner loop is peeled and placed outside the new inner loop. The memory accesses inside this peeled body are virtualized to force the prefetching of the cache lines they access before the iterations of the new innerloop start.

The upper bound of the new innerloop is a function of the array access vectors, and is dynamically recomputed between iterations of the next loop level. It corresponds to the largest possible bound that guarantees safe access within the prefetched cache lines for all the memory operations. An upper bound is computed individually for each array reference by using the current reference and stride information. A stride is the address distance between two consecutive iterations of the innermost loop for one array reference. For example, if the loop index is i, the step function is $i = i + 1$ then the stride for array reference $A[2 * i + 1]$ is $stride_A = \&A[2 * (i + 1) + 1] - \&A[2 * i + 1]$. Let us note the set of array references within the inner loop with $\Re$, denote with $Aref_k$ the k-th array access within the loop and with $Safebound(addr, stride)$ the runtime function used to calculate the safe bound for each array reference. The safe upper bound UB for the transformed innermost loop is the minimum between all the individual bounds for each array reference $Aref_k \in \Re$.

$$UB = Min(\{Safebound(\&Aref_k, stride_{Aref_k}))|Aref_k \in \Re\})$$

In figure 8 we show a simple example before and after the program transformations outlined previously. In the figure the for loop is broken into a doubly nested loop. On each iteration of the outer loop we first access the next location of both the `A` and `B` arrays so that the page containing them is guaranteed to be present in the local SRAM. Then we calculate how many more iterations can be run until the next page boundary, and the result is stored in the variable `end`. The inner loop is then run until it reaches iteration `end`, at which point we assume that a new page needs to be fetched. The outer loop then fetches the next page, and so on.

4 Experiments

This section contains the evaluation of the software caching system. The experiments are performed on a cycle-accurate simulator of the Raw microprocessor. The IO nodes are implemented as auxiliary Raw tiles executing a specially written program to handle the off-chip memory.

The simulator uses MIPS R2000 as the processing element on each tile. Both network and resource contention are faithfully simulated. The characteristics of the applications studied are presented in table 3. The cost of a load for a hit-case is presented in table 4. In all the experiments shown we have 32KB SRAM memory per tile.

```
int A[2000];
int B[4000];
main(){
  int i;
  A[0] = 0;
  B[0] = 0;
  for (i = 1; i < 2000; i++){
    A[i] = A[i-1] + i;
    B[2*i] = B[2*i -
2] + A[i];
  }
}
```

(a) Input program

```
int A[2000];
int B[4000];
extern int main()
{
   int i,int end, min_k;
   *A = 0;
   *B = 0;
   i = 1;
   if (i < 2000) {
      do {
         A[i] = A[i - 1] + i;            // will be virtualized
         B[2 * i] = B[2 * i - 2] + A[i]; // will be virtualized
         min_k = Safebound(&A[i], 4);
         min_k = min(min_k, Safebound(&A[i - 1], 4));
         min_k = min(min_k, Safebound(&B[2 * i], 8));
         min_k = min(min_k, Safebound(&B[2 * i - 2], 8));
         min_k = min(min_k, Safebound(&A[i], 4));
         end = min(min_k + i, 2000);
         i = i + 1;
         if (i < end) {
            do {   // predictable accesses
               *(int *)(((unsigned int)&A[i] & 32767u) + _MEM) =
                  *(int *)(((unsigned int)&A[i - 1] & 32767u) + _MEM) + i;
               *(int *)(((unsigned int)&B[2 * i] & 32767u) + _MEM) =
                  *(int *)(((unsigned int)&B[2 * i - 2] & 32767u) + _MEM) +
                  *(int *)(((unsigned int)&A[i] & 32767u) + _MEM);
               i = i + 1;
            } while (i < end);
         }
      } while (i < 2000);
   }
}
```

(b) Transformed program

Figure 8: A simple example for the Hot Pages predictable compiler transformation.

Benchmark	Type	Source	Description
Jacobi	Dense Matrix	Raw benchmarks	Jacobi
MxM	Dense Matrix	Nasa7:Spec92	Matrix multiplication
Life	Dense Matrix	Raw	Conway's game of life
Vpenta	Dense Matrix	Nasa7	Inverts 3 pentadiagonals
Cholesky	Dense Matrix	Nasa7	Cholesky decomposition
Moldyn	Irregular, scientific	Chaos	Molecular dynamics
Adpcm	Streaming	Mediabench	Audio compression
Median-filter	Streaming	-	Filter to find median of 9 points
Sor	Regular	-	Five point stencil relaxation

Table 3: Benchmark characteristics.

The results we show include the Hot Pages likely optimization and preliminary results with the Hot Pages predictable scheme on one tile.

The performance of several applications with the Hot Pages likely system using a software direct mapped cache is presented in figure9. We show five bars for each application. The first bar corresponds to an unoptimized version of the software handler. The second bar corresponds to optimized inlined hit handlers, selective virtualization but without the Hot Pages likely analysis. The third column gives the performance for a complete Hot Pages likely system. The fourth bar is the estimated performance of a hardware direct mapped cache with 32word cache-line assuming no memory and network contention effects but fixed off-chip access costs. Although the hardware scheme has a few optimistic assumptions we prefer not to account for contention and synchronization costs to make the comparison easier to grasp. The fifth bar is the simulated performance of a system with infinite memory per tile.

As we can observe, the overhead of the Hot Pages system varies within the range of 18% to 120% compared with the HW caching scheme. For five of the applications the overhead is close or less than 50%. Interestingly, the applications with larger overheads are the simplest ones (Life, Jacobi, MxM) impacted by the inability of the Raw compiler to optimize the modified versions as efficiently as the original programs. The explanation is that many of the compiler optimizations performed are within basic blocks affected by the more involved control structures after the software caching passes. However, these applications are also those that are easiest to fully analyze, and our preliminary results with the Hot Pages predictable scheme show that we will be able to improve on their performance. Figure 10 shows that applications such as Jacobi, SOR and our syntetic application have a performance similar to what a hardware caching system can achieve. We are currently working on integrating the Hot Pages predictable program transformations with the parallelization passes so that we could run parallel multiple tile versions with this technique.

The Hot Pages likely system has an impressive average predicition rate of 90% (see table 5) eliminating the need of cache-tag lookups in most of the cases. Used to reduce the number of accesses to the tag memory it could be applied to save large portion of the power required for data caching.

The Hot Pages system is fully customizable. We can select caching strategies, compiler techniques and cache line size based on application requirements. This is advantageus as many applications have very different requirements that often cannot be exploited in a fixed hardware caching system. In figure 13 we show two applications with completely opposite cache-line requirements. In figure 11 we show that the software column associative scheme eliminates large portion of conflict misses when needed. Compared with fixed hardware schemes, Hot Pages can provide more associativity when needed, having the ability of exceeding the performance of hardware solutions.

All of the applications studied show very good *scalability* with added processors. In figure 12 and figure 14 we show the performance of two applications representing a regular and an irregular application. The scalability of several other applications is presented in [12]. The first bar within each processor configuration measures the runtime of the application using the base software caching solution without any optimization of the software overheads but with selective virtualization. The second bar shows the execution time with software caching using a hand-optimized version of the hit-case handler implemented with Machsuif (similar to MIPS assembler) and selective virtualization. The third bar within each configuration shows the execution time by also including the Hot Pages likely compiler technique. The fifth bar is a measured runtime assuming 1 cycle load and store costs and no misses, corresponding to an ideal solution with infinite memory. The fourth bar is constructed by adding the total cost of cache misses to the infinite memory case, and assuming the same page size and off-chip access cost as in the software solution. The numbers at the top of the bars are scaled to the fourth bar or the execution time of the hardware cache with finite memory. The problem size of Jacobi is chosen in such a way that the data fully fits on four tiles. This is the explanation why we get more than a factor of two speedup going from a 1 tile to a two tile configuration. From four tiles on we only get the impact of cold misses. For this problem size our software solution on 4 tiles is actually faster than the hardware solution with infinite memory on a two tile configuration. Moldyn is an irregular scientific application with both affine and non-affine array accesses. Moldyn benefits both from computational parallelism and memory parallelism, its software caching performance scales with added processors. We can also see that the Hot Pages likely optimization gives good performance improvement even for this irregular application.

5 Related Work

Our scheme is most similar to a software version of the Translation Lookaside Buffer (TLB), technique widely used to reduce the address translation overhead in modern virtual memory systems. The most important difference is that our solution does not require a dynamic lookup of TLB entries at runtime. Instead, it leverages static compile-time information to specialize memory accesses so that the right entry is accessed without runtime intervention. The page table organization used for translation in Hot Pages is similar to those used in virtual memory systems in contemporary microprocessors. A survey of six commercial memory management designs can be found in [10].

There is a large body of research related to hardware solutions for caching. Our work has been mostly influenced by research where compile-time and runtime software mechanisms were used to improve the performance.

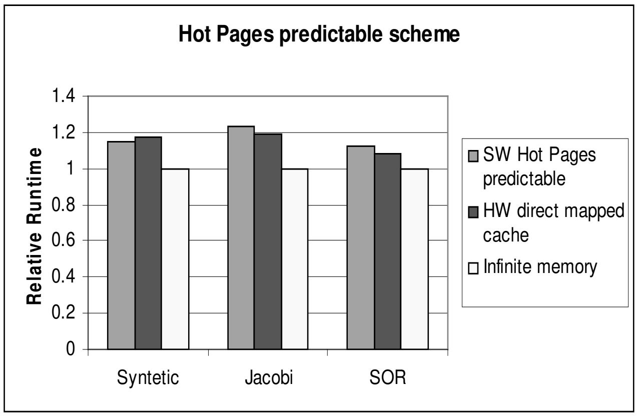

Figure 10: Hot Pages predictable system. The syntetic application shows the performance achievable in the optimal case when only one of the memory accesses within each cache line are virtualized.

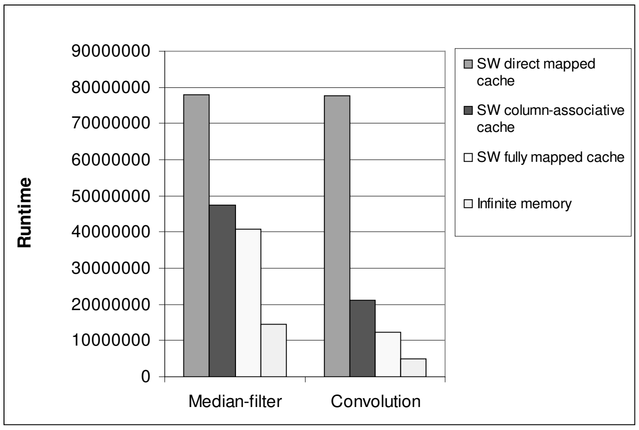

Figure 11: Example of applications with a lot of conflict misses running on a one tile system with 256 word cache-line and the Hot Pages likely system. The Hot Pages system with the two-way column-associative strategy succesfully eliminates large portion of the conflict misses. This can be seen when compared with the fully-mapped system (not practical for these problem sizes) that only has capacity misses and no conflict misses.

Method	Fully-mapped	Direct Mapped	Column Associative
Base	24	23	23
Inlined	10(9)	9(8)	9(8)
Hot Pages likely	6(5)	5(4)	5(4)
Hot Pages predictable	1	1	1

Table 4: Cost of a load for the hit-case in cycles. The "Base" method corresponds to implementing the handler in C as a runtime library procedure. The three translation mechanisms used are fully mapped, direct mapped and two-way column associative software cache (cost is shown for primary entry). The "Inlined" version is a hand-optimized inlined handler written in Machsuif. The "Hot Pages likely" and "Hot Pages predictable" cases are the handler times for hotpage accesses. The hit-times shown in parenthesis will be achieved in the final version when the start address of the mapped SRAM will be fixed (because we could use a register plus constant offset addressing mode).

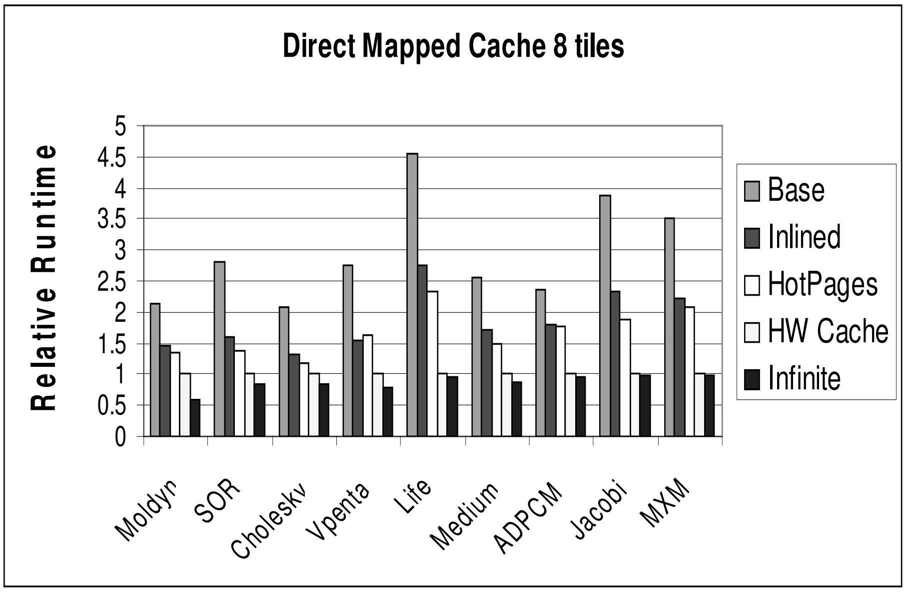

Figure 9: Runtime of several applications using a software direct mapped cache with 256words cache lines and the Hot Pages likely system. Results are normalized to a simulated hardware direct mapped cache with 32 words cache lines. Fetching a cache-line from the DRAM with hardware support is assumed to require $2 * packets * (L_{chip-boundary} + L_{on-chip}) + L_{DRAM} + CACHELINE/DRAM_{bandwidth} = 2*2(2+4)+3+32*1.5 = 75$ cycles.

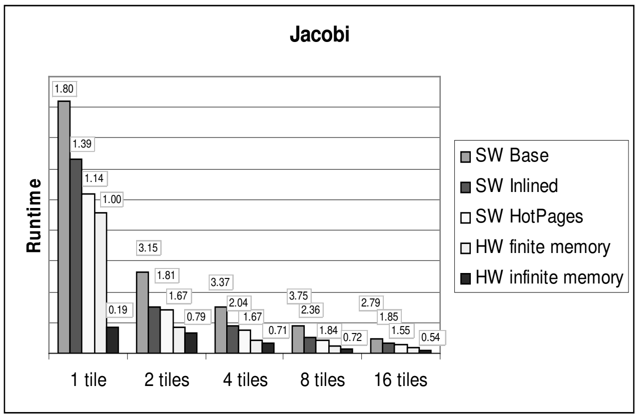

Figure 12: Runtime for different processor configurations and caching methodologies for Jacobi using the Hot Pages likely system. The numbers at the top of the bars are scaled to the HW finite memory case within each processor configuration.

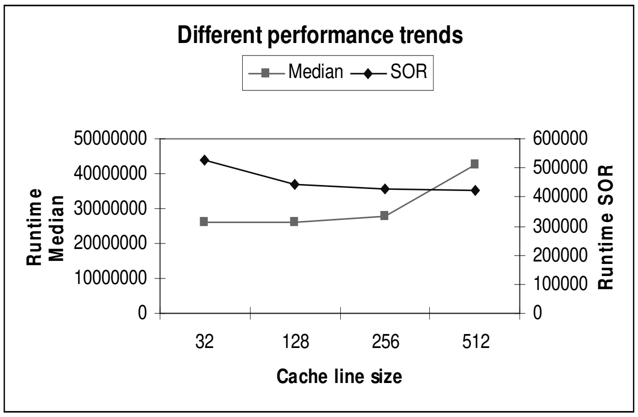

Figure 13: Performance trends in function of cache-line size. Note, that these applications have opposing requirements in cache line sizes that give optimal performance. One of the advantages of a software solution is its ability to individually select the cache line size that gives the best performance. Results derived using the Hot Pages likely caching system.

Application	Total virtualized	Misspredicted	% eliminated tag-checks
Jacobi	19392	505	97.3
MxM	12928	73	99.4
Cholsky	25440	4502	82.3
Life	2516	54	97.8
Moldyn	243740	11487	95.2
Adpcm	51202	420	99.1
Sor	14131	1024	92.7
Vpenta	6868	3734(2321)	45.6 (66.2)
MedianF	410018	21416	94.7

Table 5: Percentage of tag-checks (page table lookups) eliminated with the Hot Pages likely technique. In every experiment we allow maximum 3 hotpages. The second result shown for Vpenta corresponds to increasing the number of hotpages allowed to 8. The total percentage of tag checks eliminated is actually higher because a large portion of memory accesses use the local stack directly.

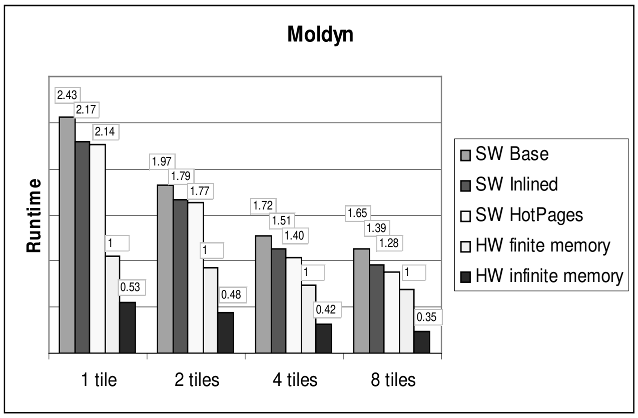

Figure 14: Runtime for different processor configurations and caching methodologies for Moldyn using the Hot Pages likely system. The numbers at the top of the bars are scaled to the HW finite memory case within each processor configuration.

The Shasta system [15] supports fine granularity sharing in software on cluster of computers with physically distributed memory. Although Shasta leveraged compile-time information to some degree, its transformations were based on a binary modification tool called ATOM [16] on a program intermediate format where much of the high-level information already has been altered. Several other systems such as Olden [5], Split-C [6] use compiler generated checks to support a global address space in the context of a parallel programming model. These systems solve a different problem, namely the sharing of the global address space in a parallel execution environment. The Hot Pages system in contrast, does the mapping between the on-chip memories and off-chip DRAM, and it does it in such a way that no overlapping can exist between the tile virtual memories. In addition, the programming model for Raw is sequential – the compiler has full control over parallelization and distribution of program data across tiles.

The Application-level Virtual Memory [7] is an attempt to move components of the virtual memory system into application level so that specialized memory management solutions can be used for each application. Example of resources that can be specialized include page table organizations and page size similar to our solution. However, the key difference between Hot Pages and AVM is that Hot Pages implements specialization of resource management based on static compile-time information.

The Softwm approach for software address translation proposed in [9] implements address translation in software similar to our system but without leveraging compile-time information. Although it obtains low overhead for virtualization, the approach taken is mainly applicable in the case of virtually tagged and addressed caches where address translation is not on the critical execution path.

The software-enforced fault isolation or sandboxing [17] is a software approach to implementing fault isolation within a single address space. The key idea is to insert two instructions before each unsafe store or jump instruction. Sandboxing does not catch illegal addresses, but it prevents them from affecting any fault domain other than the one generating the address. Although the approach taken is similarly based on modification of the program code its focus is mainly protection not virtualization.

6 Conclusions

This paper presented a fully automated software solution for data caching. Key contributions of this system are novel techniques that leverage compile-time information. Our two techniques, the *Hot Pages likely* and the *Hot Pages predictable* systems cover both applications with statically analyzable access patterns and applications without full predictability guarantees. Hot Pages provides a new foundation for caching and a safety net enabling many other cache optimzations. We presented simulation results for a variety of applications running with Hot Pages on a prototype Raw system. Our results are very encouraging for several reasons. The compile-time techniques used in our system significantly reduce the impact of the software overheads on the execution time, many of our applications performing close to hardware solutions. Our preliminary results using Hot Pages predictable techniques show that excellent performance is achievable for certain classes of applications. The Hot Pages likely system gives a performance within factor of two from hardware solutions with similar or less than 50% overhead for half of the applications. We expect to further reduce the virtualization overhead by combining the Hot Pages predictable and the Hot Pages likely schemes. Hot Pages eliminates an impressive 90% of the cache-tag lookups and could be applied to reduce the power required for data caching. Finally, our solution is fully adaptive and successfully exploits both memory and instruction-level parallelism, providing a parallel performance that scales with added processors.

References

[1] A. Agarwal and S. Pudar. Column Associative Caches. In *In Proceedings of the 20th Annual International Symposium on*

Computer Architecture 1993, pages 179–190, San Diego, CA, May 1993. ACM. Also as MIT/LCS TM-489, January 1993.

[2] K. Asanovic. *Vector Microprocessors*. PhD thesis, University of Berkeley, Department of Electrical and Computer Engineering, 1998.

[3] R. Barua, W. Lee, S. Amarasinghe, and A. Agarwal. Memory Bank Disambiguation using Modulo Unrolling for Raw Machines. In *Proceedings of the ACM/IEEE Fifth Int'l Conference on High Performance Computing(HIPC)*, Dec 1998. Also http://www.cag.lcs.mit.edu/raw/.

[4] R. Barua, W. Lee, S. Amarasinghe, and A. Agarwal. Maps: A Compiler-Managed Memory System for Raw Machines. In *Proceedings of the 26th International Symposium on Computer Architecture*, Atlanta, GA, May 1999.

[5] M. Carlisle. Olden: Parallelizing Programs with Dynamic Data Structures on Distributed-Memory Machines, June 1996.

[6] D. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel Programming in Split-C. *Proceedings of Supercomputing*, 1993.

[7] Dawson Engler and Sandeep Gupta and Frans Kaashoek. AVM: Application-Level Virtual Memory. In *Proceedings of the Fifth Hot Topics in Operating Systems*, 1995.

[8] B. Greenwald. A Technique for Compilation to Exposed Memory Hierarchy, September 1999.

[9] B. Jacob and T. Mudge. Software-Managed Address Translation. In *Proceedings of the 24nd Annual International Symposium on Computer Architecture*, 1997.

[10] B. Jacob and T. Mudge. Virtual Memory in Contemporary Microprocessors. *IEEE Micro*, Aug. 1998.

[11] 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.

[12] C. A. Moritz, M. Frank, W. Lee, and S. Amarasinghe. Hot pages: Software caching for raw microprocessors. (LCS-TM-599), Sept 1999.

[13] C. A. Moritz, D. Yeung, and A. Agarwal. Exploring Performance-Cost Optimal Designs of Raw Microprocessors. In *In Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, Napa, CA, April 1998. IEEE Computer Society.

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

[15] D. J. Scales, K. Gharachorloo, and C. 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.

[16] A. Srivastava and A. Eustace. Atom: A system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN 94 Conference on Programming Language Design and Implementation*, pages 196–205, 1994.

[17] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient Software-Based Fault Isolation. In *Proc. Fourteenth ACM Symposium on Operating System Principles*, pages 203–216, Dec. 1993.

[18] 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.