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 based solution for managing on-chip data on the MIT Raw Machine, a scalable, parallel, microprocessor architecture. This software system transparently manages the mapping between the program address space and available on-chip memory. 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 design and test, and provides flexibility for customized solutions.

The challenge for this kind of software based approach is to balance the tradeoffs between the added software overheads against opportunities provided by a memory management scheme specialized for each application. Our technique, called *Hot Pages*, combines both compile-time and runtime techniques to reduce the software overheads. The Hot Pages system optimizes the translation code for the hit case and caches translated virtual page descriptions likely to be reused for nearby memory references. We use pointer analysis to identify program memory references that can reuse a translated virtual page description. This allows us to specialize the code for each memory reference, reducing the cost of performing translations, and eliminating it entirely in some cases. The framework also provides additional opportunities for optimization because the cost of sophisticated memory management schemes can be relegated to translation misses. We present simulation results for a variety of applications running with Hot Pages on a prototype Raw system.

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. 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 and low cost. 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. Implementing a software caching solution would be a feasible alternative, also because 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 microprocessor 8% of the power is consumed by the TLB management and 18% by the data cache [1]. A combined software-hardware memory management solution, where the hardware is switched off for accesses that can be handled efficiently in software, could be used to optimize the power consumption required for memory management.

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

A typical Raw system might include a Raw microprocessor [15, 10], 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 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. Unfortunately, this is not an easy tradeoff to manage. Cache management schemes that provide better hit-rate or require less data memory for translation purposes add significant software overheads for the common hit-case. To benefit from more sophisticated software memory management solutions we need to develop new techniques that reduce or mask the overheads added to the common case hit-path.

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 optimizing the software handler for the hit case and by caching translated virtual page (is a contiguous address range in DRAM that is mapped into the SRAM of one tile) descriptions likely to be reused for nearby memory references. A page in the SRAM is similar to a cache line in hardware caching systems. Address translation is the

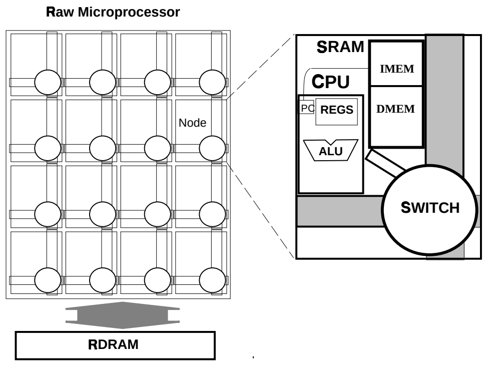


Figure 1: Raw system composition. A typical Raw system includes a Raw microprocessor coupled with off-chip DRAM and stream IO devices. Each Raw tile contains a simple RISC-like processor, an SRAM memory for instructions and data, and a switch. The tiles are interconnected in a 2D mesh network that is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

process of mapping a program address on a Raw tile to an SRAM address. A novel aspect of our system is the compile-time analysis that identifies if a program memory reference can reuse a previous virtual page translation. We call such a virtual page a *hot page* for two reasons: (1) the compiler guarantees that its translation is saved into registers for fast access, and (2) the compiler identifies other memory accesses that likely can reuse that translation. The address translation for a hot page reference is customized at compile-time. It uses directly the specific hot page page translation (saved in registers). References mapped to hot pages require no table lookup or hashing during runtime. Note, that several hot page 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.

The specific contribution of this paper include:

- The description and evaluation of a fully-automated software only solution for caching.
- Novel compile-time techniques used to reduce/mask the overhead of software schemes.

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 scheme. 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.

2.1 Runtime Mechanisms

The runtime system can be divided into three components, (1) address translation, (2) page table organizations, and (3) 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 in several banks and the virtual address spaces seen by individual tiles are distinct, simultaneous DRAM accesses can be supported.

Page Table Organizations The mapping of virtual addresses into physical SRAM addresses is organized into page tables, which are a 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 because of 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).

Replacement Mechanism Our system currently implements a circular FIFO replacement policy. 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 hot pages are non-replaceable and are ignored during 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 [3] and [9].

The base Raw compiler generates code assuming infinite memory per tile. Our hotpage system removes this assumption by imple-

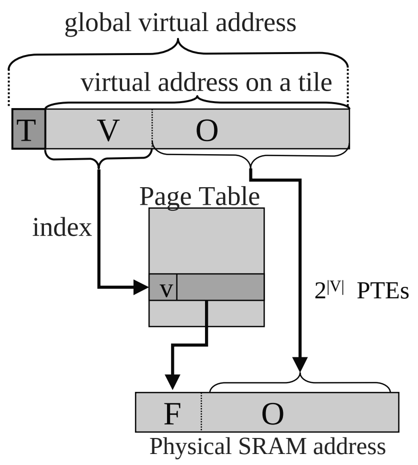


Figure 2: Address translation using a fully-mapped single-level page table. The main advantage of this organization is its relatively low overhead for a page table entry lookup. The disadvantage is the memory space required for the entries which is reflecting the size of the virtual address space. Notation: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

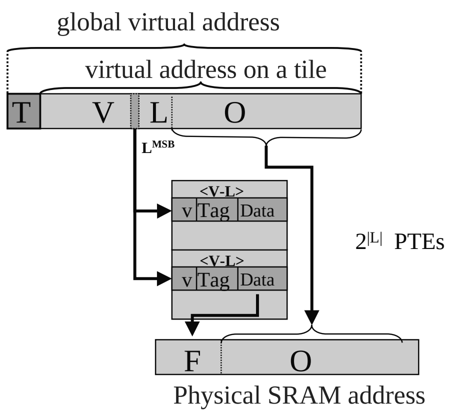


Figure 3: Address translation using an inverted page table with two-way set associative mapping. This organization has low memory requirement for page table entries, proportional with physical SRAM memory , but has a high translation overhead also because of the necessary tag checking and the potential 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.

menting 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 hotpage 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 hot page analysis phase which divides memory references into groups called *hot page sets*. All references inside a hot page set will essentially use the translation saved for a specific virtual page called a *hot page*. All the memory accesses that are cacheable are changed with library procedure calls in the virtualization pass. Note, that both the analysis phases and the virtualization 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 is an important pass as it eliminates the need for runtime checks in case of multiple hot page 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.

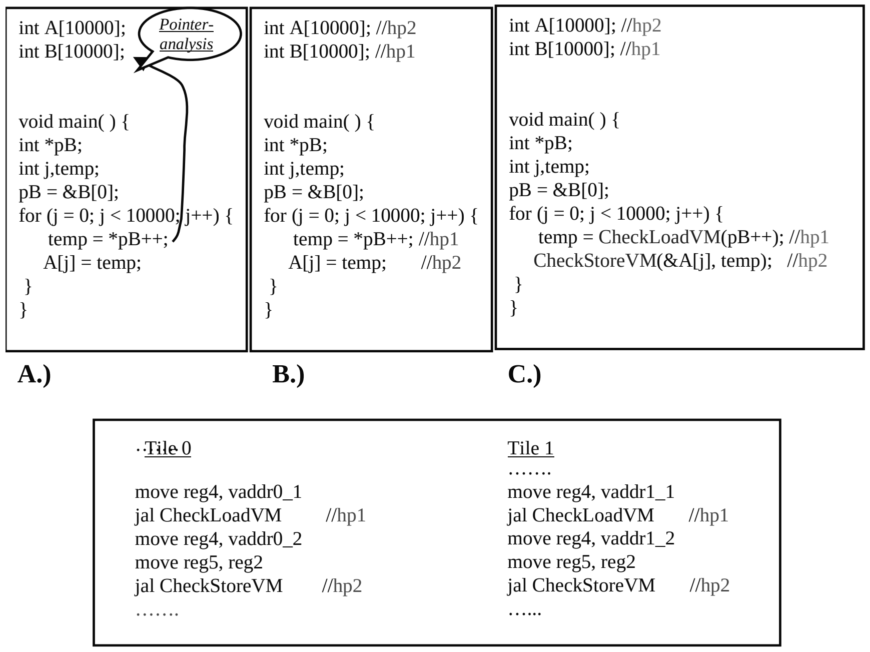
3.1 Pointer Analysis

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 [11]. 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



```
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;
 }
}
```

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 hot page 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 hot page set annotations (i.e., *hp1, hp2*). Note that the hot page sets on Tile0 and Tile1 are different.

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*. 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 [3] and [2].

3.3 Hot Page Analysis

In this section we describe the compiler analysis for Hot Pages. Our analysis is performed on the Stanford University Intermediate Format (SUIF). Hot Page 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 is the process of mapping a program address on a Raw tile to an SRAM address. 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) alleviate 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 *hot page sets*. Hot page sets are determined based on their location sets (information given by pointer analysis) in the memory. Each hot page set contains references that can likely reuse the address translation saved for a specific virtual page called *hot page*. The compiler algorithm has two phases. First, it finds data objects such as arrays and structures and map these objects to hot page sets. Then, it traverses the control-flow graph of the program and maps memory accesses to existing hot page 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 than it can likely reuse the address translation saved for the hot page 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 ac-

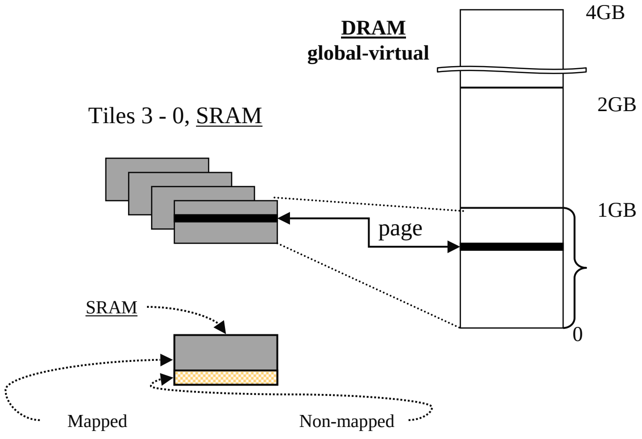


Figure 4: Global and Tile Virtual Memory. Note that the memory disambiguation and data distribution phases in the compiler are guaranteeing that the virtual spaces on different tiles are non-overlapped. For example each tile in the figure has 1GB of virtual address space. Tile0's virtual address space 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 memory is used to provide fast access (1 cycle) for local stack accesses.

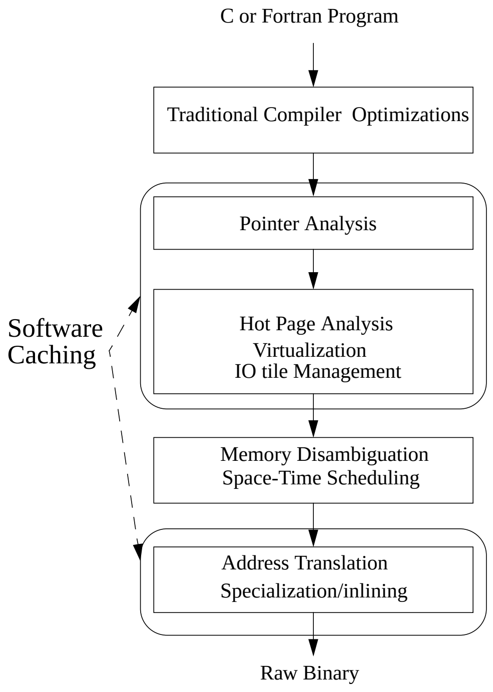


Figure 5: Structure of the Raw Compiler with Software Caching

cess 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 hot page. This runtime check is not necessary for the predictable case.

Each hot page has two pieces of information assigned to it: a virtual page number and a physical frame, both saved into registers. A hot page 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 hot page set. To reduce register pressure our compiler currently uses three (number determined empirically) hot page sets, and reuses these for different location sets in different regions in the program.

3.4 Virtualization

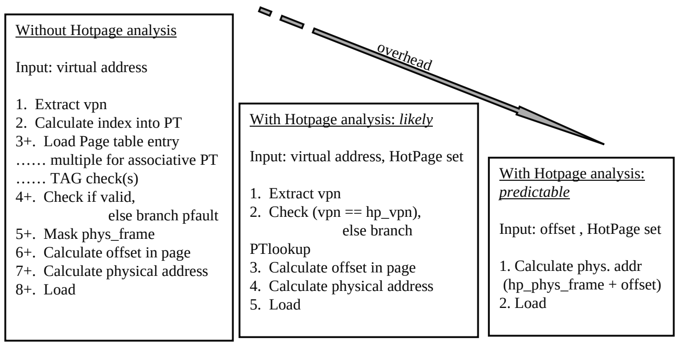
The compiler determines which accesses should be cached in the virtualization phase. If an access is virtualized than it is substituted with a procedure call into the virtual memory system. The compiler can decide not to virtualize an access and map it to a non-cached (non-mapped) SRAM memory area. These 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.

3.5 Specialization

The address translation for hot page references is customized at compile-time directly for the specific hot page page description (translation). No extra table lookup or hashing for accessing a hot page 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 hot page 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 hot page 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.

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 1. The cost of a load for a hit-case is presented in table 2. The results for each application are presented in separate graphs, see the figures 8 to 13. Each graph contains the runtime information for different tile configurations and caching techniques used. In all the experiments shown we have 32KB SRAM memory per tile, 1KB page sizes (cache lines), single-level fully mapped page table organization and modified circular FIFO replacement. The results we show include the Hot Pages likely optimization, We are currently working on incor-



Without Hotpage analysis
Input: virtual address
1. Extract vpn
2. Calculate index into PT
3+. Load Page table entry
...... multiple for associative PT
...... TAG check(s)
4+. Check if valid, else branch pfault
5+. Mask phys_frame
6+. Calculate offset in page
7+. Calculate physical address
8+. Load
overhead
With Hotpage analysis: likely
Input: virtual address, HotPage set
1. Extract vpn
2. Check (vpn == hp_vpn), else branch PTlookup
3. Calculate offset in page
4. Calculate physical address
5. Load
With Hotpage analysis: predictable
Input: offset , HotPage set
1. Calculate phys. addr (hp_phys_frame + offset)
2. Load

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 hot page set, hp_phys_frame= register containing the Physical Page Frame for a hot page set. Several hot page sets can coexist at the same time each using separate hp_phys_frame and hp_vpn registers.

porating the Hot Pages predictable analysis. We expect the performance to improve with the addition of the Hot Pages predictable compiler techniques. We are also working on adding the necessary support to be able to have a more exact breakdown of costs.

The first bar within each processor configuration measures the runtime of the application using the base software caching solution without any optimization. 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). The third bar within each configuration shows the execution time by also including the Hot Pages likely compiler optimization. The fifth bar is a measured runtime assuming 1 cycle load and store costs and no misses, corresponding to an ideal HW 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. This bar does not include the synchronization delays caused by dependent instructions across tiles where some of the instructions are delayed due to cache misses and off-chip accesses.

The numbers at the top of the bars are scaled to the execution time of the fourth bar or the hardware case with finite memory. Our intention with this is to show the parallel execution time degradation of various software schemes compared to an architecture with 1 cycle loads and stores, no synchronization costs caused by cache misses, and finite memory per tile.

The first application we studied was Jacobi. Our best software solution which is with the Hot Pages likely optimizations performs between 14-84% worse than the hardware finite memory solution. The problem size 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. Jacobi scales nicely with added processors.

Matrix multiply has similar performance behavior. The overhead of our best scheme with the Hot Pages likely optimization compared to the hardware finite memory case is between 51-82%. The parallel performance scales with added processors. Both Jacobi and MxM have almost perfect hit rate for hot pages.

Method	Load
Base	30
Inlined	13
Hot Pages likely	5
Hot Pages predictable	2

Table 2: 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 translation used is based on a one-level fully mapped page table organization. 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 hot page accesses.

Life is an application where synchronization delays due to the sequentialized cache misses impact on the performance. Although we get speedup, the speedup is not as good as in the infinite memory case. We can see that the cost of cache-misses only affects the performance by approximatively factor of two on 8 tiles (see the difference between the HW infinite memory runtime and the HW finite memory runtime) so the rest of the performance degradation is due to synchronization delays not accounted in the HW finite memory bar. Of course this is a pessimistic performance comparison for the software schemes, as a real hardware solution with finite memory would incur synchronization delays too. We are currently investigating ideas on reducing the synchronization delays by integrating the static instruction scheduling tighter with the software caching passes.

The performance of Vpenta on one tile is effected by a lower hit-rate due to its large problem size relative to the SRAM size. Increasing the number of tiles reduces the replacement misses and also parallelizes the off-chip accesses. The performance improves with added processors, however we can observe synchronization delays on larger configurations due to dependencies between instruction streams on different tiles.

Cholesky is an application where our Hot Pages likely scheme is slower than the other software schemes on one tile. The explanation is that Cholesky has triple-nested loops with multiple array accesses. The arrays are not indexed with the innermost loops index causing most of our hot page predictions to fail. Because our replacement algorithm does not replace virtual pages that are *hot* we even get lower hit-rate than in the other software schemes. The performance of this scheme improves however on larger configurations where because of smaller loop bounds and larger on-chip memory the hit rate of hot page accesses is significantly improved. On 8 tiles the parallel performance of the Hot Pages likely scheme is the best between the software solutions, being only 40% slower than the runtime would be with the hardware finite memory case.

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. Its parallel performance using Hot Pages is only 28% slower than the hardware finite memory performance on 8 tiles. We can also see that the Hot Pages likely optimization gives good performance improvement even for this irregular application. Note, that we expect the performance gap to increase between the Hot Pages solution and the other software solutions if more sophisticated address translation schemes (i.e. based on inverted page tables with associative mapping) and replacement mechanisms are used.

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

Table 1: Benchmark characteristics.

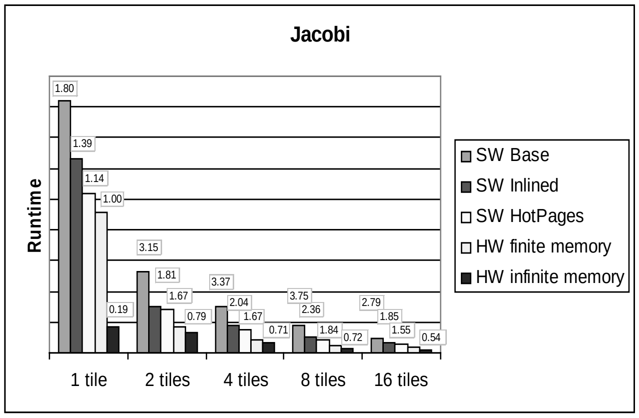


Figure 8: Runtime for different processor configurations and caching methodologies for Jacobi. The numbers at the top of the bars are scaled to the HW finite memory case within each processor configuration.

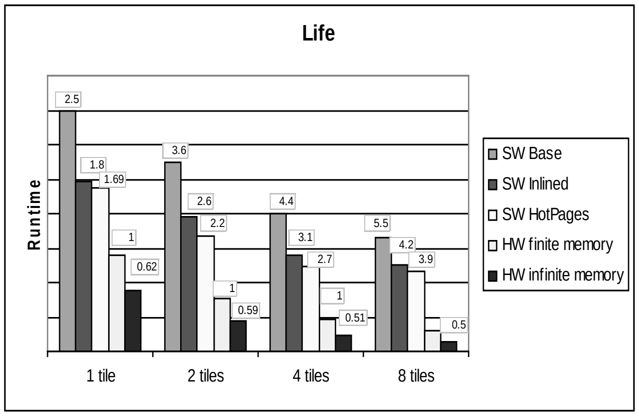


Figure 10: Runtime for different processor configurations and caching methodologies for Life. The numbers at the top of the bars are scaled to the HW finite memory case within each processor configuration.

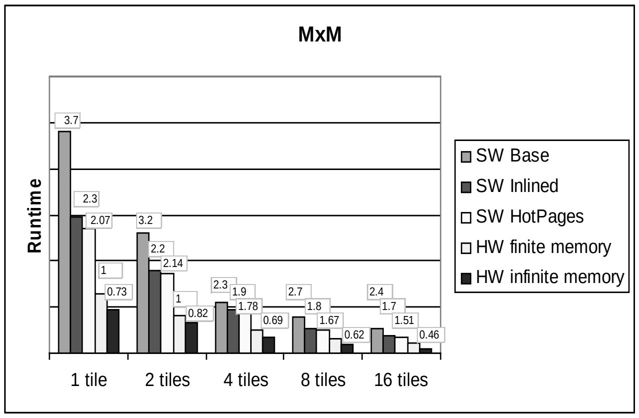


Figure 9: Runtime for different processor configurations and caching methodologies for MXM. The numbers at the top of the bars are scaled to the HW finite memory case within each processor configuration.

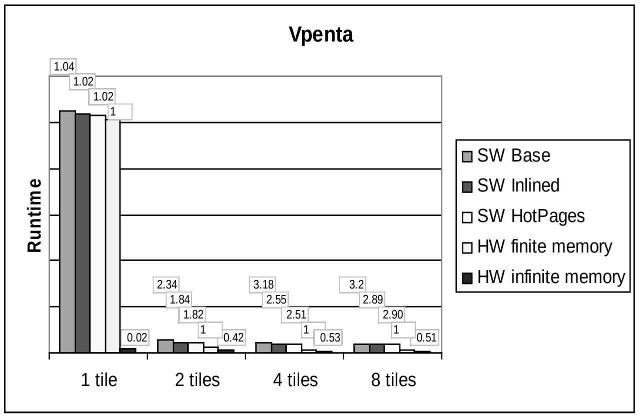


Figure 11: Runtime for different processor configurations and caching methodologies for Vpenta. The numbers at the top of the bars are scaled to the HW finite memory case within each processor configuration.

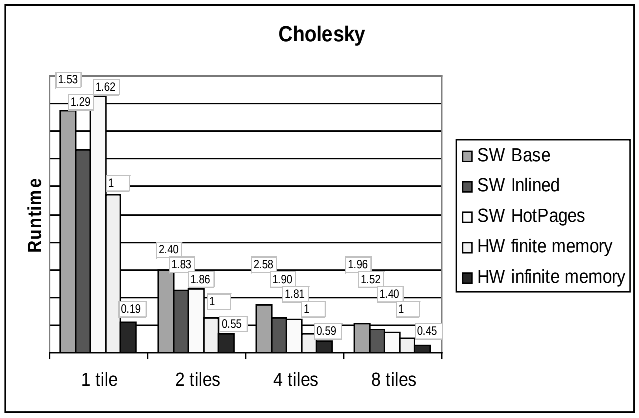


Figure 12: Runtime for different processor configurations and caching methodologies for Cholesky. The numbers at the top of the bars are scaled to the HW finite memory case within each processor configuration.

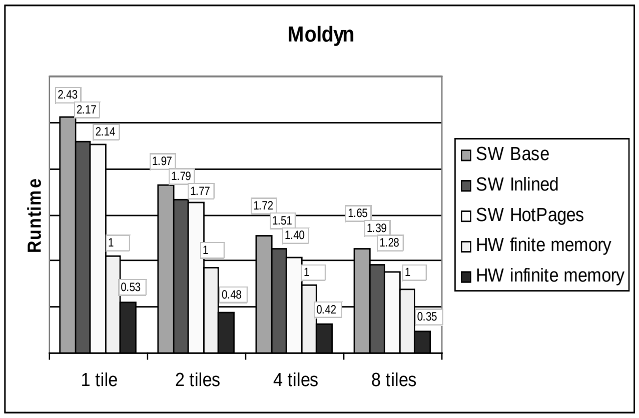


Figure 13: Runtime for different processor configurations and caching methodologies for Moldyn. The numbers at the top of the bars are scaled to the HW finite memory case within each processor configuration.

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 [8].

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.

The Shasta system [12] 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 [13] on a program intermediate format where much of the high-level information already has been altered. Several other systems such as Olden [4], Split-C [5] 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 [6] 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 [7] 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 [14] 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-only solution for caching. Key contribution of our scheme is leveraging compile-time information to reduce and eliminate the overheads of software schemes. Besides lowering software overheads, the Hot Pages solution provides opportunities for further optimizations as it can potentially mask the overhead of more sophisticated memory management schemes. We presented simulation results for a variety of applications running with Hot Pages on a prototype Raw system. Our results are encouraging of several reasons. First, the compile-time techniques used in our system significantly reduce the impact of the software overheads on the execution time, most of our applications performing within factor of two of the ideal hardware solution. Second, our solution successfully exploits both memory and instruction-level parallelism, providing a parallel performance that scales with added processors. Finally, the virtualization overhead of the system can be further reduced with the Hot Pages *predictable* implementation and with the use of software prefetching.

7 Acknowledgements

The research is funded by DARPA contract #DABT63-96-C-0036. We are grateful to Radu Rugina, Martin Rinard for the help in using the pointer analysis package, Anant Agarwal, Krste Asanovich, Benjamin Greenwald, Jonathan Babb, Mike Taylor, Rajeev Barua, Mark Stephensen and other members of the Raw group for many relevant discussions related to Raw and caching.

References

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

[2] 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/.

[3] 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.

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

[5] 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.

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

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

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.