FlexCache: A Framework for Flexible Compiler Generated Data Caching

Csaba Andras Moritz[1], Matthew I. Frank[2], and Saman Amarasinghe[2]

[1] University of Massachusetts,
Electrical and Computer Engineering,
Amherst, Ma 01002
andras@ecs.umass.edu
[2] Massachusetts Institute of Technology,
Laboratory for Computer Science,
Cambridge, Ma 02139
{mfrank,saman}@lcs.mit.edu

Abstract
This paper describes our work in progress on FlexCache, a framework for flexible, compiler generated data caching. FlexCache substitutes the tag-memory and cache controller hardware with a compiler managed tag-like data structure, address translation, and tag-check. This allows the division of the data-array into separately controlled partitions, allows the selection of cache line sizes, the support of highly associative mappings, and the selection of various replacement policies on a per program basis.

FlexCache leverages compile-time static information to selectively virtualize memory, to eliminate cache-tag accesses, and to guide the replacement of conflicting cache lines. For the applications studied the FlexCache compiler techniques eliminate in average more than 90% of the cache-tag lookups, enabling the support of highly associative caching schemes. Even without any hardware support, FlexCache can outperform fixed hardware caches by improving caching effectiveness, eliminating mapping conflicts, and eliminating the cache pollution caused by register spills. The beauty of FlexCache is that its core techniques could be augmented with additional software techniques and/or hardware support (i.e., special instructions) to look into optimizing data caching in areas such as low-power, real-time systems, and high-performance microprocessors.

1 Introduction

Several recent project proposals focus on issues related to memory resources, as memory resources still represent the most important bottleneck in computing devices, in respect to power-consumption, performance, and area occupied [4]. Because of rapidly changing requirements for optimal designs, these projects emphasize on memory systems that are reconfigurable or optimizable to some extent. Good example is Smart Memories [6] that successfully maps architectures with different memory requirements to a low-level computing fabric. The

Berkeley IRAM project [9] improves memory latency by designing a processor in DRAM process and includes the DRAM memory on-chip. The MIT Raw project [10] focuses instead on compiler technology to control low-level hardware resources to optimize the performance of memory accesses. The FlexRAM project [5] moves computation closer to the DRAM in an attempt to reduce memory latency for memory bound applications by combining processing and DRAM on the same chip. The Active Pages system [7] shifts data-intensive computations to the memory system by combining FPGAs with DRAM. A new hardware cache is shown in [8] that divides the SRAM storage into several partitions and use it as prefetch buffer, compiler controlled memory, or lookup buffer. This design is shown to benefit media applications with up to 20% performance improvement. Digital Signal Processors such as the Texas Instruments TMS320C62XX added hardware support to use SRAM data storage as both cache and main memory. To cope with the long DRAM latencies in future microprocessors a fully associative secondary cache with software replacement/control is suggested in [3]. A software replacement algorithm is shown to achieve miss-rate reductions from 8% to 85% compared to a 4-way LRU.

This paper describes FlexCache, a framework for flexible, compiler generated data caching. FlexCache takes the flexibility of the above mentioned memory systems one step further. Instead of building upon conventional caching architectures, FlexCache builds upon a flexible software platform with the possibility of adding hardware resources as needed. Hardware support can be added in form of new instructions to reduce the overhead of the software schemes, but without altering the flexibility and programmability provided.

FlexCache substitutes the tag-memory and the cache controller with a compiler managed tag-like data structure, address translation, and tag-checks. It allows the division of the data-array into separately controlled partitions, the selection of cache line sizes, the support of highly associative mappings, and the selection of various replacement policies. Because FlexCache leverages static information available during compilation, it can improve caching efficiency compared to conventional caches. Eliminating the hardware for managing cache-tags and providing direct access to fast SRAM can also simplify the pipeline and reduce design overhead. FlexCache incorporates new compiler techniques called Hot Pages, succesfully eliminating tag-checks, making possible the design of highly associative caches. The FlexCache approach is different from many other compiler systems that aimed compiler optimizations at reducing the impact of memory latency in the context of conventional hardware caches.

FlexCache uses four main techniques in its base implementation to reduce overheads during the hit case. First, it can avoid caching when not necessary by mapping (at compile time) some memory accesses directly into fast SRAM memory. Example of such accesses are register spills that otherwise pollute caches and can cause significant performance degradation [2]. Second, it uses for the first time compiler techniques enabled by global pointer analysis to identify and speculatively reuse previously calculated cache mappings at runtime. In our software implementation this optimization reduces the software overhead for the hit

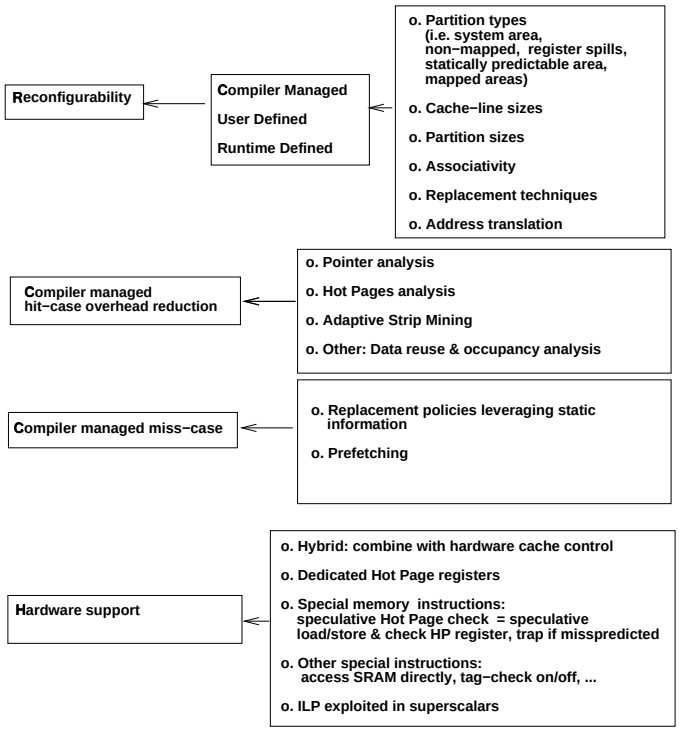

Fig. 1. Overview of the FlexCache framework. FlexCache can combine compiler analysis, user input, or runtime profiling for cache configuration selection. New compiler techniques are proposed to reduce software overheads in software managed caching, to manage the miss-case efficiently, and to eliminate tag-checks. Hardware support can be added in form of new speculative memory instructions, direct access to the cache data-array, and in form of dedicated Hot Page registers. Additionally, a FlexCache proposes to incorporate hardware cache-control providing the ability to handle some memory accesses in conventional manner.

case to four instructions per load (in a pure software implementation running on a single-issue processor), independent of the sophistication of the underlying caching strategy. The four instruction overhead can be reduced to two instructions on multiple issue processors, and can be completely eliminated with special Instruction Set Architecture (ISA) extensions. Third, for dense array accesses the system uses *adaptive strip mining* to reduce the number of tag checks from once per memory access to once per cache line. Finally, the FlexCache system allows the user to select cache-line size, associativity and replacement policy for each program, reducing the miss rate.

Additional compiler techniques could be developed to automate the selection of these parameters. This paper mainly focuses on identifying the critical compo-

nents of the FlexCache system and leaves out the compiler techniques required to automate the configuration selection process.

An overview of the FlexCache framework is presented in Figure 1. A FlexCache based cache can combine compiler analysis, user input, or runtime profiling for optimal cache configuration selection. FlexCache incorporates new compiler techniques to reduce software overheads in software managed caching, to manage the miss-case efficiently (i.e., by leveraging compiler knowledge we can design application specific replacement policies), and to eliminate cache tag-checks. Hardware support can be added in form of new speculative memory instructions, direct access to the cache data-array, support for access to different type of memory partitions, and in form of dedicated Hot Page registers. Additionally, FlexCache proposes to incorporate hardware cache-control, in a hybrid cache solution, providing the ability to handle some memory accesses in conventional way (i.e., same manner as in a hardware cache).

Our preliminary results demonstrate that compile time analysis can eliminate a large portion (90% in average for the applications studied) of the cache-tag lookups. Even without any hardware support, FlexCache can outperform fixed hardware caches by improving caching effectiveness, eliminating mapping conflicts, and eliminating the cache pollution caused by register spills. Because all the main FlexCache features are software based, a FlexCache solution is application specific and can be easily retargeted to focus on different design aspects such as low power consumption, high performance, or high predictability of memory accesses. FlexCache can also be the right solution to add data caching for FPGA based systems if it is incorporated in a silicon compilation system such as described in [1].

The remainder of this paper is organized as follows. Section 2 and 3 describes the components of the FlexCache system. Section 4 presents suggestions for hardware support. Section 5 gives our preliminary experimental results. Section 6 concludes the paper.

2 FlexCache Runtime System

This paper assumes an architecture with a local SRAM memory and an external large DRAM. Address translation is the mapping of program addresses into SRAM addresses. This mapping can be implemented by using a table lookup similar to the page table lookup used in virtual memory systems. The mapping is done at a *line* granularity. A line is a contiguous address range in both SRAM and program memory. Each translation table entry contains both a tag and a translation to a physical SRAM address.

The translation overhead if done in software varies for different table organizations, but requires at minimum, (1) calculating the line number from the program address, (2) calculating an address into the translation table, based on the line number, (3) loading the tag, (4) comparing the line number with the tag, (5) calculating the offset within the line, (6) loading the translation, (7) calculating the actual physical SRAM address from the translation and offset, (8)

actually loading the data from SRAM. Associative caching schemes may require multiple table lookups and tag compares (steps 2-4). Hardware cache management schemes provide special hardware to perform all of the work for steps 1-7, usually in a single cycle. Section 3 describes our compiler optimizations to reduce the software overhead. The Hot Pages optimization eliminates steps 2, 3 and 6. ISA extensions implementing the remaining operations could eliminate all the remaining overhead, while keeping (i.e., not altering) the flexibility of the compiler based solution.

When the FlexCache system detects a cache miss (the software tag-check fails) it invokes a miss handler. Table entries that correspond to Hot Pages (cache-lines identified as hot during compile-time) are non-replaceable and are ignored during replacement. In the direct mapped cache there is only one line that can be replaced. If we map hot pages 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 hot page accesses will not access lines that currently map to hot pages and therefore we can move the mapping of hot pages outside the main mapping table. This improves the hit-case as cache lines predicted statically to be hot can only be replaced with other (static) hot lines.

As the gap between processing speeds and external DRAM latencies is rapidly widening, with DRAM latencies reaching thousands of processing cycles, managing cache misses in software becomes a feasible approach.

3 FlexCache Compiler

Traditionally, the compiler generates code assuming infinite memory. The compiler component of the FlexCache system removes this assumption by implementing caching in software. Figure 2 shows the flow of the augmented FlexCache compiler which includes the new phases required for caching.

This section shortly describes the compiler components of the FlexCache system. First, it describes pointer analysis, an analysis technique used to determine the location set list of each memory reference. Next, it describes the Hot Page analysis phase which divides memory references into groups called *hot page sets*. All references inside a hot page set will use the same register allocated translation, called a *hot page*, an important overhead reduction optimization. Figure 3 introduces an example which illustrates the steps the compiler performs for software managed caching.

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 FlexCache system, the analysis is used to guide the placement of data and for the hot page optimization. There are many variants of pointer analysis algorithms, that mainly differ in the precision at which memory references are disambiguated. The FlexCache system is based on a most precise type of

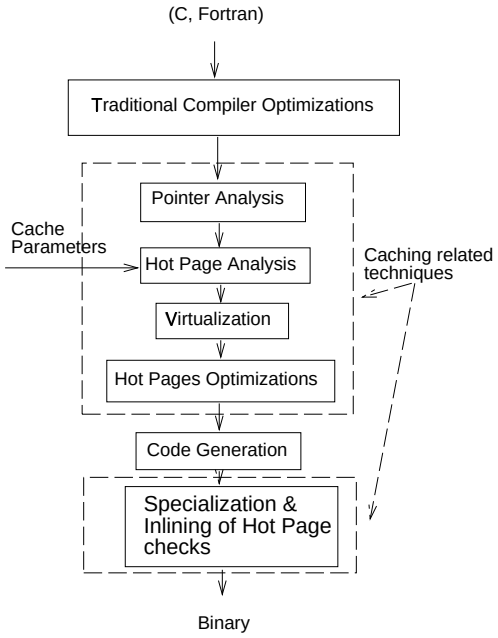

Fig. 2. Structure of the FlexCache Compiler

pointer analysis that is both flow-sensitive (i.e., takes control-flow into account) and context-sensitive (i.e., location sets may differ based on calling context).

However, the FlexCache approach do not require this type of precision for correctness of execution, less precise but faster pointer analysis techniques could have been used, perhaps with somewhat less static prediction accuracy for cache accesses. Details about the pointer analysis used can be found in [11, 12].

Hot Page Analysis

The Hot Page analysis pass 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.

Our technique leverages static information about the locality of accesses, to implement a fast 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 memory. Each hot page set contains references that can likely reuse the address translation saved for a specific memory line called a *hot page*.

The compiler algorithm has two phases. First, it finds data objects such as arrays and structures and maps 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

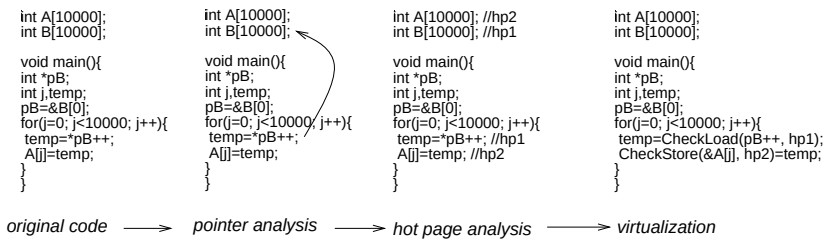

Fig. 3. An example of how FlexCache implements software caching in the compiler: A.) Pointer analysis is used to determine the location sets of memory references, for example *pB* has the same set as *B*, B.) the *Hot Pages* analysis annotates memory references into hot page sets *hp1, hp2*, C.) the *virtualization* pass changes the memory references with procedure calls. The address translation *specialization* pass in the compiler back-end inlines *specialized* code for *CheckLoad, CheckStore*. This specialization is controlled by the hot page set annotations (i.e., *hp1, hp2*).

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 hot page set assigned to that array. Note that several location sets can be *hot* at the same time.

Each Hot Page has two pieces of information assigned to it: a program line number (the tag) and a physical frame, both saved into registers. A successful translation done through a Hot Page register will add only 4 cycles of overhead in a full software implementation: one cycle to extract the line number from the address, one cycle to compare to the tag stored in the `hp_vpn` register, one cycle to extract the line offset from the program address, and a final cycle to add the offset to the translated SRAM address held in the `hp_phys_frame` register. The next section will show additional techniques based on architectural extensions that eliminates the 4 cycles.

If the Hot Page tag check fails, then a procedure call is made to the slower software runtime tag check routine, which takes about 23 cycles in our system. Alternatively, a hybrid system would rely on the hardware cache check in case of misspredicted Hot Page access.

As mentioned earlier, the Hot Page check itself could be speed up by implementing the hot page check in the instruction set of the processor.

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. The compiler can decide not to virtualize an access and map it directly to an unmapped portion of SRAM. These accesses are local and only cost 1 cycle (*i.e.,* as fast as having a hardware cache). The compiler maps scalar register spills to this area. Larger stack mapped objects are handled through the software caching system to avoid overloading the unmapped portion of SRAM.

The address translation for Hot Page references is customized at compile-time for the specific Hot Page page description (translation). No extra table lookup

or hashing for accessing a Hot Page description is required at runtime, as the code is generated with the right registers at compile-time. If the FlexCache ISA support is in place then the compiler can generate these memory instructions directly.

4 Hardware Support

In an environment where performance is critical there is a possibility of combining conventional cache control with a software FlexCache. The support that is needed is the separation of memory accesses at compile time into hardware and software managed. This support can be easily added based on the location set information available at compile time.

Another option to improve performance is to add hardware support to reduce the software overheads in software caching. Perhaps the most promising idea is extension of the ISA with special instructions to reduce the latency of the Hot Pages tag-checks. The Hot Pages compiler techniques reduces the overhead in a software managed cache to four operations per memory access, by register promoting (i.e., storing into registers) address translations of memory accesses that are likely to result in cache hits. As we show in the next section, for several benchmarks the compiler successfully predicts which accesses can benefit from register promotion, achieving an average prediction accuracy above 90%. A Hot Pages based load/store operation can be implemented as a speculative load/store instruction. This approach reduces a Hot Pages based memory access to one instruction (depending on correct prediction in the compiler).

With this support, a fully software managed cache partition has the higher predictability provided by the software techniques (as the selected cache parameters for the partition can be application specific), and maintains the low hit-case latency of a hardware based cache. Figure 4 shows three different implementations. The exact sequence of MIPS assembly code is shown for illustrative purposes only, to give an idea of the overhead in a software cache partition. If the Hot Pages load is executed on a single-issue processor the additional overhead compared to a regular load (see instructions before *lw*) is 4 instructions. Executing the same code on a superscalar processor and exploiting ILP (Instruction Level Parallelism) reduces this overhead to two instructions. If architecture support is provided, i.e., if implemented as a special memory instruction, *hplw* [**?**], then a Hot Pages memory operation can run at the speed of a regular load.

Figure 5 shows the data-path for the hplw instruction. It illustrates that the additional hardware required to implement this instruction is minimal, and that misspeculation can be detected early in the Execution stage of the pipeline.

A hybrid FlexCache, that incorporates the architectural support briefly described above, maintains the added flexibility and improved efficiency provided by the software techniques, in addition to a hit-case latency similar to that of conventional hardware based implementations. In a hybrid solution we can use hardware cache partitions for memory accesses that are not predictable or analyzable at compile-time (or memory accesses that are infrequent and hard to

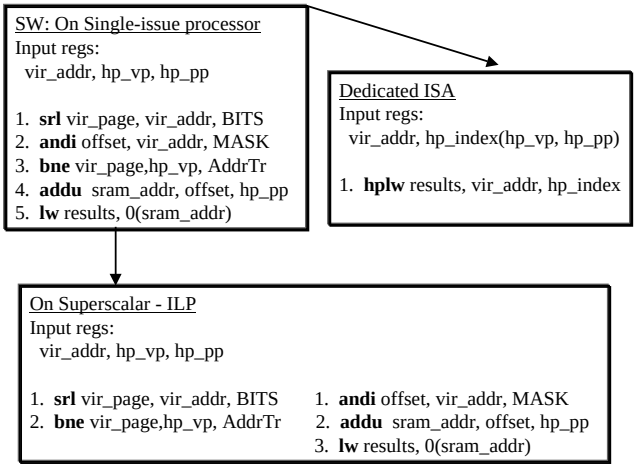

Fig. 4. Three different possible implementations of Hot Pages loads using a MIPS ISA. First, we have a load with 4 additional instructions overhead on single-issue processors. On a superscalar processor this overhead is reduced to two instruction per load because of the ILP in the check code. With the ISA extension the overhead is completely eliminated.

analyze). Example of such accesses are non-affine array accesses and references to complex data structures using pointers. More predictable references are dealt with using compiler managed memory areas or software caching, with added architectural support.

5 Experiments

We implemented the FlexCache system on a single-issue 5 stage pipeline microprocessor similar to MIPS R4000. A two level memory hierarchy based on a local SRAM and an off-chip DRAM is simulated. A local on-chip SRAM memory is attached to the processor. A large external RDRAM memory is necessary to solve large problems that exceed the size of the on-chip memory. The experiments are performed on a cycle-level simulator.

We have found that the FlexCache system has an average dynamic memory access prediction rate of 90% for the applications studied (see Table 1), eliminating the need of cache-tag lookups (or address translations) in most cases.

The FlexCache system is fully customizable that can benefit many applications. Applications have very different requirements that often cannot be exploited in a fixed hardware caching system. In Figure 7 we show two applications with very different cache-line requirements. Execution times vary as much as 70%.

In Figure 6 we show that the software column associative scheme can eliminate many conflict misses that are produced in the direct mapped case. Com-

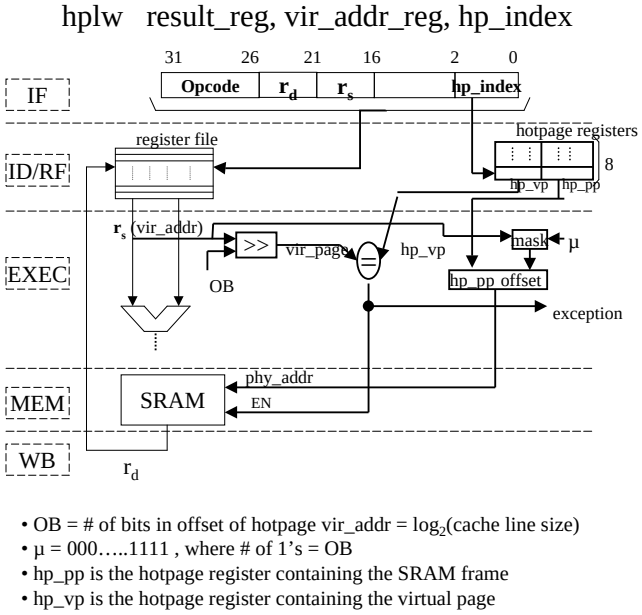

Fig. 5. Data-path of the *hplw* instruction. Note that the speculative addressing of the cache can be done with minimal hardware support. The HotPages registers containing the HotPages translations are used to reduce register pressure. A misspredict ion is handled as an exception and it is detected early in the Execution stage of the processor pipeline.

pared with a fixed direct mapped hardware caching scheme, FlexCache can provide more associativity (i.e., because it is software managed) when needed. The Convolution program is three times faster with the software FlexCache system compared to a system using a fixed direct mapped hardware cache.

6 Conclusions and Future Work

This paper presented FlexCache, a framework for flexible, compiler managed data caching.

For the applications studied the FlexCache techniques eliminate in average more than 90% of the cache-tag lookups, enabling the support of highly associative caching schemes while providing full flexibility. We have shown that even without any hardware support, a FlexCache based system can outperform fixed hardware caches by improving caching effectiveness.

We are currently working on adding hardware support in a hybrid FlexCache, by extending the ISA with specialized memory instructions that speculatively access local memory using address mappings from the Hot Page registers, on building low-power *hardware* managed caches that expose just enough of the

Application	Total Cache Access.	Mispredicted	% optimized
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	2321	66.2
MedianF	410018	21416	94.7

Table 1. Percentage of tag-checks (translation table lookups) optimized with the Hot Pages technique in a FlexCache system. The total percentage of tag checks eliminated is actually higher because a large portion of memory accesses use the local stack directly.

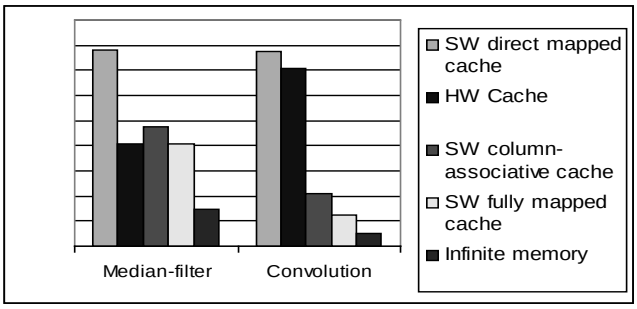

Fig. 6. The effect of cache associativity for programs with many conflict misses. Bars shown are normalized to the execution time of the software direct mapped case. For each program, the bars, left to right correspond to (1) a software FlexCache with direct mapping, optimized with Hot Pages, and 256 word cache-lines, (2) A hardware direct mapped cache with 32 word (with 256 word size the performance is worse!) cache-lines, (3) a software FlexCache with 2-way column-associative mapping, optimized with Hot Pages, and 256 word cache-lines, (4) a software FlexCache with fully mapped table, optimized with Hot Pages and 256 word cache lines, and (5) an ideal memory.

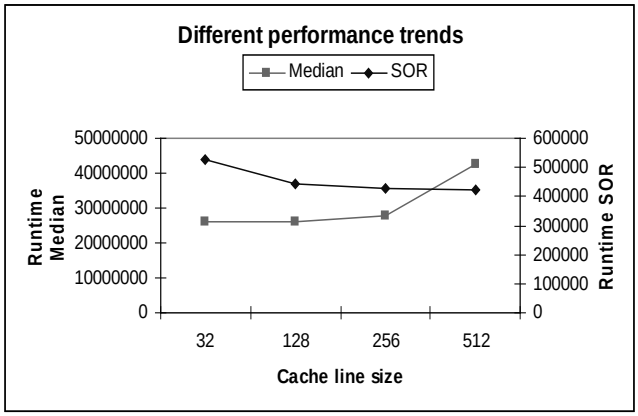

Fig. 7. Different line sizes are better for different programs. Runtime for the Median-filter and SOR programs with a software FlexCache using 2-way column associative mapping and optimized with Hot Pages. Median-filter does better with smaller cache lines while the SOR program does better with a larger line size.

cache interface to the compiler, such that the FlexCache techniques can be used to reduce tag-array accesses.

References

1. J. Babb, M. Rinard, C. A. Moritz, M. Frank, W. Lee, R. Barua, and S. Amarasinghe. . In *Parallelizing Applications into Silicon*, Napa, CA, April 1999. IEEE Computer Society.
2. K. D. Cooper and T. J. Harvey. Compiler-Controlled Memory. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–11, San Jose, CA, Oct. 3–7 1998.
3. S. K. R. Erik G. Hallnor. A Fully Associative Software-Managed Cache Design. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, Vancouver, June 2000. IEEE Computer Society.
4. R. Fromm, D. Patterson, K. Asanovic, A. Brown, J. Golbus, B. Gribstad, K. Keeton, C. Kozyrakis, D. Martin, S. Perissakis, R. Thomas, N. Treuhaft, K. Yelick, T. Anderson, and K. Wawrzynek. Intelligent RAM (IRAM). In *IRAM tutorial, ASP-DAC98*. IEEE Computer Society, Februari 1998.
5. Y. Kang, W. Huang, S.-M. Yoo, D. Keen, Z. Ge, V. Lam, P. Pattnaik, and J. Torrelas. FlexRAM: Toward an Advanced Intelligent Memory System. In *Proceedings of International Conference on Computer Design (ICCD)*, Oct 1999.
6. K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz. Smart Memories: A Modular Reconfigurable Architecture. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, Vancouver, June 2000. IEEE Computer Society.
7. M. Oskin, F. T. Chong, and T. Sherwood. Active Pages: A Computation Model for Intelligent Memory. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, Barcelona, June 1998. IEEE Computer Society.

8. N. P. J. Parthasarathy Ranganathan, Sarita Adve. Reconfigurable Caches and Their Application to Media Processing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, Vancouver, June 2000. IEEE Computer Society.
9. D. Patterson, T. Andersson, N. Cardwell, R. Fromm, K. Keeton, and C. K. and. A Case for Intelligent RAM: IRAM. *In IEEE Micro*, April 1997.
10. S. A. Rajeev Barua, Walter Lee. Maps: A Compiler-Managed Memory System for Raw Machines. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, Atlanta, June 1999. 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. R. P. Wilson and M. S. Lam. Efficient Context-Sensitive Pointer Analysis for C Programs. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 1–12, La Jolla, CA, June 18–21, 1995.