# Runtime Biased Pointer Reuse Analysis and Its Application to Energy Efficiency

Yao Guo, Saurabh Chheda, Csaba Andras Moritz

Department of Electrical and Computer Engineering
University of Massachusetts, Amherst, MA 01003
{yaoguo, schheda, andras}@ecs.umass.edu

**Abstract.** Compiler-enabled memory systems have been successful in reducing chip energy consumption. A major challenge lies in their applicability in the context of complex pointer-intensive programs. State-of-the-art high precision pointer analysis techniques have limitations when applied to such programs, and therefore have restricted use. This paper describes runtime biased pointer reuse analysis to capture the behavior of pointers in programs of arbitrary complexity. The proposed technique is runtime biased and speculative in the sense that the possible targets for each pointer access are statically predicted based on the likelihood of their occurrence at runtime, rather than conservative static analysis alone. This idea implemented as a flow-sensitive dataflow analysis enables high precision in capturing pointer behavior, reduces complexity, and extends the approach to arbitrary programs. Besides memory accesses with good reuse/locality, the technique identifies irregular accesses that typically result in energy and performance penalties when managed statically. The approach is validated in the context of a compiler managed memory system targeting energy efficiency. On a suite of pointer-intensive benchmarks, the techniques increase the fraction of memory accesses that can be mapped statically to energy efficient memory access paths by 7-72%, giving a 4-31% additional L1 data cache energy reduction.

## 1 Introduction

The memory system, including caches, consumes a significant fraction of the total system power. For example, the caches and translation look-aside buffers (TLB) combined consume 23% of the total power in the Alpha 21264 [7], and the caches alone use 42% of the power in the StrongARM 110 [8]. Recent studies have proposed compiler-enabled cache designs [2, 12, 14] to improve cache performance as well as energy consumption. A major challenge, however, is their applicability when dealing with complex pointer-intensive programs. This paper presents a new approach to deal with complex pointer-intensive programs in such schemes based on the idea of runtime biased pointer reuse analysis. In addition to compiler-enabled memory systems, applications such as compiler-based prefetching, software-based memory dependence speculation, and parallelization, could also significantly benefit from the techniques presented in this paper.

Many researchers have focused on program locality/reuse analysis for array-based memory accesses [9, 15, 16]. In general, array accesses are more regular than pointer-based memory accesses because arrays are normally accessed sequentially while pointers typically have more complicated behavior. Array based accesses are also relatively easy to deal with as type information is available to guide the analysis.

Intensive use of pointers makes however program analysis difficult since a pointer may point to different locations during execution time; the set of all locations a pointer can access at runtime is typically referred to as the *location set*. This difficulty is further accentuated in the context of large and/or complex programs. For example, more precise dataflow-based implementations of pointer analysis have limitations (e.g., often cannot complete analysis) when used for large programs or when special constructs such as pointer based calls, recursion, or library calls are found in the program. The less precise alias analysis techniques (e.g., those that are flow-insensitive) have lower complexities but don't provide precise enough static information about pointer location sets.

Our objective is to develop new techniques to capture pointer behavior that can be used to analyze complex applications with no restrictions, while providing good precision. The idea is to determine pointer behavior by capturing the frequent locations for each pointer rather than all the locations as conservative analysis would do. Predicted pointer reuse is therefore runtime biased and speculative in the sense that the possible targets for each pointer access are statically predicted/speculated based on the likelihood of their occurrence at run-time. The approach enables lower complexity and possibly higher precision analysis than traditional dataflow based approaches because locations predicted to be infrequently accessed are not considered as possible targets. The approach is applicable in all architecture optimizations that use some kind of compiler-exposed speculation hardware and when absolute correctness of static information leveraged is not necessary. This includes for example compiler managed energy-aware memory systems, compiler managed prefetching, and speculative parallelization and synchronization - these applications by their design would benefit from precise memory behavior information and would tolerate occasional incorrect static control information.

This paper shows the application of the proposed pointer techniques to an energy-efficient compiler-enabled memory management system published previously, called Cool-Mem [2]. The Cool-Mem architecture achieves energy reduction by implementing energy efficient statically managed access paths in addition to the conventional ones. The compiler decides which path is used based on static information extracted. For accesses that reuse the same cache line, cache mapping information is maintained to help eliminate redundancy in cache disambiguation. Whenever the compiler can correctly channel data memory accesses to the static access path, significant energy reduction is achieved; the statically managed access path does not need Tag access and associative lookup in RAM-Tag caches, and Tag access in CAM-Tag caches.

We show that the Cool-Mem architecture, if extended with our techniques, is able to handle pointer based accesses and achieve up to 30% additional energy savings in the L1 data cache.

The rest of this paper is structured as follows. Section 2 presents the runtime biased compiler analysis techniques, including pointer analysis, distance analysis, and reuse analysis algorithms. Following this, Sect. 3 provides an overview of the compiler-enabled memory framework used for simulation and Sect. 4 shows the experimental framework. Finally, Sect. 5 gives the experimental results gathered through simulation, and we conclude with Sect. 6.

## 2 Compiler Analysis

The runtime biased (RB) pointer reuse analysis can be separated into a series of three steps: RB pointer analysis, RB distance analysis, and RB reuse analysis.

*RB Pointer Analysis* is first applied in order to gather basic pointer information needed to predict pointer access patterns. A flow-sensitive dataflow scheme is used in our implementation. Flow-sensitive analysis maintains high precision (i.e., the location set of each pointer access is determined in a flow-sensitive manner even if based on the same variable). Our analysis is guided by reevaluating, at each pointer dereference point, the (likely) runtime frequency of each location a pointer can point to. For example, possible locations that are from definitions in outer loop-nests are marked or not included when the pointer is dereferenced in inner loops and if at least one new location has been defined in the inner loop. Conventional analysis would not distinguish between these locations.

Precise conventional pointer analysis usually requires that the program includes all its source codes, for all the procedures, including static libraries. Otherwise, the analysis cannot be performed. Precise conventional pointer analysis is often used in program optimizations where conservative assumption must be made - any speculation could result in incorrect execution.

In contrast, our approach does not require the same type of strict correctness. If the behavior of a specific pointer cannot be inferred precisely, we can often speculate or just ignore its effect. For example, if a points-to relation (or location) cannot be inferred statically, we speculatively consider only the other locations gathered in the pointer's location set. We mark the location as undefined. When assigning location sets for the same pointer at a later point in the CFG, one could safely ignore/remove the *undefined location* in the set, if the probability of the pointer accessing that location, at the new program point, is low (e.g., less than 25% in our case).

The main steps of our RB pointer analysis algorithm are as follows: (1) build a control-flow graph (CFG) of the computation, (2) analyze each basic block in the CFG gradually building a PTG, (3) at the beginning of each basic block merge location set information from previous basic blocks, (4) mark locations in the location sets that are unlikely to occur at runtime, at the current program point, as less frequent, (5) mark undefined locations or point-to relations; (6)

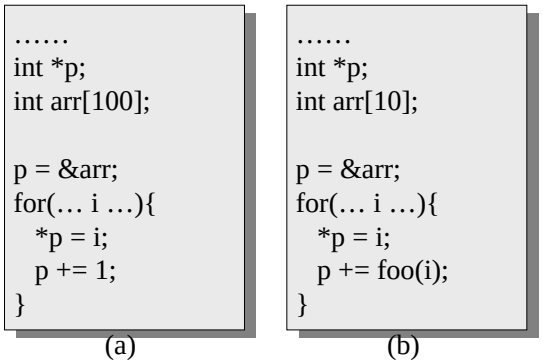


**Fig. 1.** Distance analysis examples: (a) static stride (b) variable stride

repeat steps 2-5 until the PTG graph does not change (i.e., full convergence is reached) or until the allowed number of iterations are reached.

Library calls that may modify pointer values and for which source codes are not available are currently speculatively ignored. If a pointer is passed in as an argument, its location set after the call-point in the caller procedure will be marked as speculative, signaling that the location set of the pointer might be incomplete after the call. In none of the programs we have analyzed we have found library modified pointer behavior to be a considerable factor in gathering precise pointer reuse information.

*RB Distance Analysis* gathers stride information for pointers changing across loop iterations. This stride information is used to predict pointer-based memory access patterns, and speculation is performed whenever the stride is not fixed. As strides could change in function of the paths taken in the Control-Flow Graph (CFG) of the loop body, only the most likely strides (based on static branch prediction) are considered.

In the example shown in Fig. 1(a), the value of pointer $p$ changes after each iteration. In general, there are two ways to deal with this situation if implemented as part of pointer analysis. Each element in the array structure could be treated as a different location, or, another approach would be to treat the whole array *arr* as a single location. The former is too complicated for compiler analysis while the latter is not precise enough.

In our approach, as shown in Fig. 1(a), we first find the initial location for $p$. Then, when we find out that $p$ is changing for each iteration, we calculate the distance (stride) between the current location and the location after modification. If the distance is constant, we will use both the initial location and distance to describe the behavior of the pointer.

Extracting stride information is not always easy. In Fig. 1(a), we can easily calculate that the stride for pointer $p$ is 4 bytes. However, for the example in Fig. 1(b), the stride for pointer $p$ is variable since we do not know what value

procedure *foo()* will return. In this case, we can use speculation based on static information related to the location set to estimate the stride. For example, the information we do know is (1) $p$ points to array *arr* and (2) the size of array *arr* is small. Based on this information, we can speculate that the stride of $p$ is small although we do not know the exact number.

Another example of stride prediction, as also mentioned earlier, is ignoring strides that are less likely to occur at runtime based on static branch prediction. Clearly, depending on which path is executed at runtime the stride of a pointer might change across loop iterations, as not all the possible paths leading to that pointer access are equally likely to occur.

*RB Reuse Analysis* attempts to discover those pointer accesses that have reuse, i.e., refer to the same cache line. Reuse analysis uses the information provided by the previous analyses to decide whether two pointer accesses refer to the same cache line. Based on the reuse patterns, pointer accesses are partitioned into reuse equivalence classes. Pointers in each equivalence class have a high probability of referring to the same cache line during execution and will be mapped through the static access path in the Cool-Mem system.

Reuse analysis for array-based accesses has been studied and used in [9, 15, 16]. For pointer-intensive programs, we use a classification scheme similar to theirs, but we redefine it specifically in the context of pointer-based accesses.

1. *Temporal Reuse*: This is the case when a pointer is not changing during loop iterations. This is the simplest case for loop-based accesses.
2. *Self-Spatial Reuse*: If a pointer is changing using a constant stride and the stride is small enough, two or more consecutive accesses will refer to the same cache line.
3. *Group-Spatial Reuse*: A group of pointers can share the same cache line during each loop iteration even when they do not exhibit self-spatial reuse.
4. *Simple-Spatial Reuse*: This exists between two pointers that refer to the same cache line but do not belong to any loop. Simple-spatial reuse is added as a new reuse category because we find that this situation is important for pointer-based programs although it is not as important for array-based programs. The reason for this is that array structures are typically accessed using loops, while pointer-based data structures are often accessed using recursive functions.

Pointer-based memory accesses are partitioned into different *reuse equivalence classes* based on the reuse classification and strides. A reuse equivalence relation exists between two memory accesses if one of the above mentioned reuse relations exists between them. Intuitively, each reuse equivalence class contains those pointer accesses that have a good chance to access the same cache line.

Once we have the reuse equivalence classes, we use a *reuse probability threshold* to decide which of the equivalence classes will likely have high cache line reuse. All the accesses assigned to an equivalence class with a reuse probability smaller than this threshold or not assigned to a class, will be regarded as

irregular. In our experiments, we choose the reuse threshold such that the statically estimated reuse misprediction rate is predicted to be smaller than 33% (the overall misprediction rate could be much lower depending on the mixture of equivalence classes, but could also be larger due to the speculative nature of the information this analysis is based on).

After RB reuse analysis, all the accesses which fall into one of the four reuse categories are regarded as having good reuse possibility. Pointer accesses which have bad locality and small reuse chances are identified as irregular accesses.

## 3 Application: Compiler-Managed Memory Systems

The results of run-time biased reuse analysis can be applied to general-purpose compiler-enabled cache management systems. In this paper, we replicated a compiler-enabled energy efficient cache management framework, Cool-Mem [2], and extended it by incorporating our pointer reuse analysis techniques. We will give a simple introduction of Cool-Mem architecture in this section, detail information about Cool-Mem can be found in [2].

### 3.1 Cool-Mem Memory System

Figure 2 presents an overview of the Cool-Mem memory system, with integrated static and dynamic access paths. Cool-Mem extends the conventional associative cache lookup mechanism with simpler, direct addressing modes, in a virtually tagged and indexed cache organization. This direct addressing mechanism eliminates the associative tag-checks and data-array accesses. The compiler-managed speculative direct addressing mechanisms uses the hotline registers. Static mispredictions are directed to the CAM based Tag-Cache, a structure storing cache line addresses for the most recently accessed cache lines. Tag-Cache hits also directly address the cache, and the conventional associative lookup mechanism is used only on Tag-Cache misses.

The conventional associative lookup approach requires 4 parallel tag-checks and data-array accesses(in a 4-way cache). Depending on the matching tag, one of the 4 cache lines is selected and the rest discarded. Now for sequences of accesses mapping to the same cache line, the conventional mechanism is highly redundant: the same cache line and tag match on each access. Cool-Mem reduces this redundancy by identifying at compile-time, access likely to lie in the same cache line, and mapping them speculatively through one of the hotline registers (step 1 in Fig. 2).

Different hotline compiler techniques are used to predict which cache accesses are put into which hotline registers. A simple run-time comparison (step 2) reveals if the static prediction is correct. The cache is directly accessed on correct prediction (step 3), and the hotline register updated with the new information on mis-predictions.

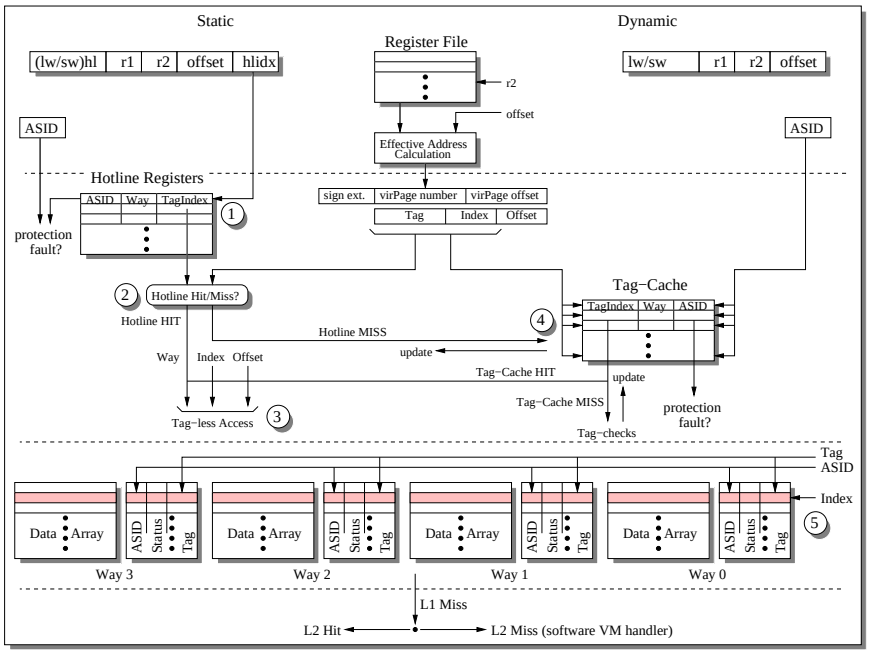


**Fig. 2.** Cool-Mem Architecture

Another energy-efficient cache access path in Cool-Mem is the CAM-based Tag-Cache. It is used both for static mis-prediction (hotline misses) and accesses not mapped through the hotline registers, i.e. dynamic accesses (step 4). Hence it serves the dual-role of complementing the compiler-mapped static accesses by storing cache-line addresses recently replaced from the hotline registers, and also saving cache energy for dynamic accesses; the cache is directly accessed on Tag-Cache hits(step 3).

Although the Tag-Cache access is very quick, we assume that the Tag-Cache, accessed on hotline misses, require another cycle, with an overall latency similar to a regular cache access. A miss in the Tag-Cache implies that we fall back to the conventional associative lookup mechanism with and additional cycle performance overhead (step 5). The Tag-Cache is also updated with new information on misses. As seen in Fig. 2, each Tag-Cache entry is exactly the same as a hotline register, and performs the same functions, but dynamically.

### 3.2 Cool-Mem Compiler

Cool-Mem compiler is responsible for identifying groups of accesses likely to map to the same cache-line, and mapping them through one of the hotline registers. Hotline passes are implemented in two different compiler techniques: (1) Optimistic Hotlines, where the compiler tries to map all accesses through the

hotline registers, and (2) Conservative Hotlines, which maps a subset of the accesses that are more regular in nature and as a result, are likely to cause fewer mis-predictions. The description of both algorithms can be found in [2].

Both the optimistic and conservative hotline approaches are not dealing with pointer variables, because pointer information is unknown without pointer alias analysis or points-to analysis. Runtime biased pointer reuse analysis results can be applied easily in the context of the Cool-Mem architecture. Simply, pointer accesses in reuse equivalence classes with reuse attributes larger than the reuse threshold are mapped to static energy-saving cache access paths. At the same time, irregular pointers identified during reuse analysis will be directed to regular cache access paths to avoid energy and performance penalties.

## 4 Experimental Framework

The SUIF [13]/Machsuif [11] suite is used as our compiler infrastructure. RB pointer and distance analysis is implemented as a SUIF pass which analyzes an intermediate SUIF file and then writes the pointer and stride information back as annotations. RB reuse analysis runs after the pointer analysis pass and writes reuse equivalence class information to the SUIF intermediate file.

The source files are first compiled into SUIF code and merged into one file. All high-level compiler analysis passes, including the pointer and reuse analysis passes, operate at this stage. The annotations from SUIF files are propagated to an *Alpha* binary file through the intermediate stages. We use the SimpleScalar [5] simulator with Wattch [4] extensions for collecting performance and energy numbers. This simulator, capable of running statically linked alpha binaries, has been modified to accommodate the Cool-Mem architecture.

We assume a 4-way in-order Alpha ISA compatible processor and 64 Kbyte 4-way set-associative L1 caches, 0.18 micron technology, and 2.0V $V_{dd}$. We account for all the introduced overheads and static mispredictions in the architecture as described in [2].

We simulated a number of benchmarks during the selection process, including SPEC 2000 [1], Olden pointer-intensive benchmark suite [10] and several benchmarks used previously by the pointer analysis community [3, 6]. We chose seven benchmarks (shown in Table 1) which contain at least 25% of pointer accesses at runtime.

## 5 Results

In this section, we show experimental results for the above benchmarks, including benchmark statistics as well as energy saving results collected using Wattch.

### 5.1 Regular vs. Irregular Pointers

Identifying those pointers which do not have good locality is important because they normally result in energy and performance penalties when managed statically. Figure 3(a) shows the percentage of irregular pointers found during static

**Table 1.** Benchmarks used in simulation results.

| Benchmark | Source | Description |
|---|---|---|
| backprop | Austin | Neural network training |
| em3d | Olden | Elect. magn. wave propag. |
| ft | Austin | Minimum spanning tree |
| ks | Austin | Graph Partition |
| 08.main | McGill | Polygon rotation |
| mcf | SPEC2000 | Combinatorial optimization |
| 09.vor | McGill | Voronoi diagrams |

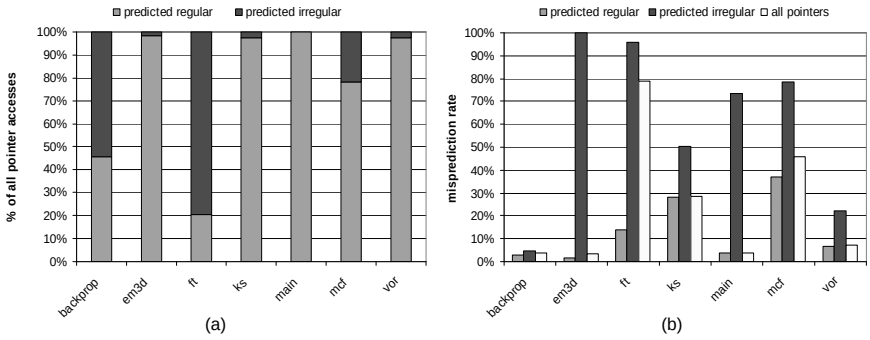


**Fig. 3.** Regular versus irregular pointers: (a) runtime percentage of statically determined regular and irregular pointers; (b) Static misprediction rates of pointer accesses when mapped to Cool-Mem's static cache access path. Misprediction occurs when a pointer that is predicted to have high reuse statically will not access the predicted cache line at runtime.

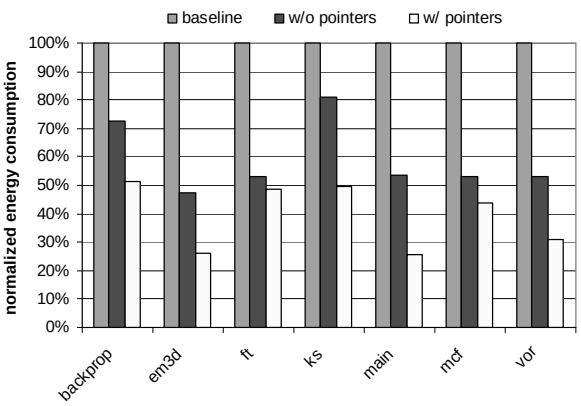


**Fig. 4.** Normalized L1 D-cache energy consumption.

compiler analysis. Different programs have a different portion of irregular pointers. In some of them, such as *main* and *em3d*, up to 99% of all the pointers are predicted as regular. Other programs like *ft* have almost 80% irregular accesses. Figure 3(b) shows the misprediction rate of the pointers predicted when mapped to the static cache access path. The misprediction rate refers to the accesses that do not point to the cache line predicted. Note that the misprediction rate for irregular pointers is very high for most of the programs. It is at least twice the misprediction rate of those pointers we identified as regular pointers. The only exception is *backprop*, which operates on a relatively small data structure, such that all the pointer accesses have very good locality. However, we can see that the misprediction rate for irregular pointers in *backprop* is still much higher than those of regular pointers.

We also show the misprediction rate for the case when all the pointer accesses are mapped through the static path. Note that the misprediction rate is significantly reduced by removing the irregular pointer accesses. For *em3d*, the overall misprediction rate is reduced by almost 50% while identifying only 1.7% of all the pointer accesses as irregular. We also identified almost 80% of all pointers as irregular in *ft*, which have a misprediction rate greater than 95%.

### 5.2 Energy Savings

Figure 4 shows the energy consumption results which are normalized to the unoptimized hardware baseline architecture. The baseline energy number, which is shown as 100%, is the first bar. The second bar shows the normalized energy consumption by applying the published Cool-Mem techniques without mapping the pointer-based accesses through the statically managed cache access path. Finally, the energy consumption number, which uses the results of our reuse analysis for pointer-based accesses, is shown last. Compared to the optimization which maps only array-based accesses, 4% to 31% extra energy reduction is

achieved on the L1 data cache energy consumption by mapping the pointer-based memory accesses that are statically predicted as regular through the statically managed cache access path.

## 6 Conclusion

Compiler-enabled cache management for pointer-intensive programs is difficult because pointer analysis is difficult and sometimes even impossible for large or complex programs. By applying the runtime biased pointer analysis techniques, we can always complete analysis for any pointer-intensive program without any constraints. The techniques proposed increase the fraction of memory accesses that can be mapped statically to energy efficient cache access paths by 7-72%, giving a 4-31% additional cache energy reduction in the L1 data cache.

Our future work includes further investigation experiments on the runtime biased pointer analysis approach and applying the analysis to other compiler-enabled techniques such as compiler-directed prefetching on pointer-intensive codes.

## References


1. The standard performance evaluation corporation, 2000. http://www.spec.org.
2. R. Ashok, S. Chheda, and C. A. Moritz. Cool-mem: Combining statically speculative memory accessing with selective address translation for energy efficiency. In *ASPLOS*, 2002.
3. T. Austin. Pointer-intensive benchmark suite, version 1.1, 1995. http://www.cs.wisc.edu/~austin/ptr-dist.html.
4. D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 83–94, Vancouver, British Columbia, June 12–14, 2000. IEEE Computer Society and ACM SIGARCH.
5. D. C. Burger and T. M. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-1997-1342, University of Wisconsin, Madison, June 1997.
6. M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 242–256, 1994.
7. M. K. Gowan, L. L. Biro, and D. B. Jackson. Power considerations in the design of the alpha 21264 microprocessor. In *Proceedings of the 1998 Conference on Design Automation (DAC-98)*, pages 726–731, Los Alamitos, CA, June 15–19 1998. ACM/IEEE.
8. J. Montanaro, R. T. Witek, K. Anne, A. J. Black, E. M. Cooper, D. W. Dobberpuhl, P. M. Donahue, J. Eno, G. W. Hoeppner, D. Kruckemyer, T. H. Lee, P. C. M. Lin, L. Madden, D. Murray, M. H. Pearce, S. Santhanam, K. J. Snyder, R. Stephany, and S. C. Thierauf. A 160-MHz, 32-b, 0.5-W CMOS RISC microprocessor. *Digital Technical Journal of Digital Equipment Corporation*, 9(1), 1997.
9. T. Mowry. *Tolerating Latency Through Software Controlled Data Prefetching*. PhD thesis, Dept. of Computer Science, Stanford University, Mar. 1994.

10. A. Rogers, M. C. Carlisle, J. H. Reppy, and L. J. Hendren. Supporting dynamic data structures on distributed-memory machines. *ACM Transactions on Programming Languages and Systems*, 17(2):233–263, Mar. 1995.
11. M. Smith. Extending suif for machine-dependent optimizations. In *Proc. First SUIF Compiler Workshop*, Jan. 1996.
12. O. S. Unsal, R. Ashok, I. Koren, C. M. Krishna, and C. A. Moritz. Cool-cache for hot multimedia. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 274–283. IEEE Computer Society, 2001.
13. R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. Lam, and J. L. Hennessy. SUIF: A parallelizing and optimizing research compiler. Technical Report CSL-TR-94-620, Computer Systems Laboratory, Stanford University, May 1994.
14. E. Witchel, S. Larsen, C. S. Ananian, and K. Asanović. Direct addressed caches for reduced power consumption. In *Proceedings of the 34th Annual International Symposium on Microarchitecture*, pages 124–133, Austin, Texas, Dec. 1–5, 2001. IEEE Computer Society TC-MICRO and ACM SIGMICRO.
15. M. E. Wolf. *Improving Locality and Parallelism in Nested Loops*. PhD thesis, Dept. of Computer Science, Stanford University, Aug. 1992.
16. M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. *SIGPLAN Notices*, 26(6):30–44, June 1991. *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation.*