Computación y Sistemas
ISSN: 1405-5546
computacion-y-sistemas@cic.ipn.mx
Instituto Politécnico Nacional
México

Alba, Marcos R. de; Kaeli, David
Exposing Instruction Level Parallelism in the Presence of Loops
Computación y Sistemas, vol. 8, núm. 1, julio-septiembre, 2004, pp. 74-85
Instituto Politécnico Nacional
Distrito Federal, México

Computación y Sistemas Vol. 8 Núm. 1, pp. 074-085
© 2004, CIC-IPN, ISSN 1405-5546, Impreso en México

RESUMEN DE TESIS DOCTORAL

Exposing Instruction Level Parallelism in the Presence of Loops

Exponiendo el Paralelismo a Nivel de Instrucciones en Presencia de Bucles

Graduated: Marcos R. de Alba
Graduated on May 1, 2004
Department of Electrical and Computer Engineering
Northeastern University
Boston MA 02115
e-mail: mdealba@ece.neu.edu

Advisor: Dr. David Kaeli
Computer Architecture Research Laboratory (NUCAR)
Northeastern University
Boston MA 02115

Abstract

In this thesis we explore how to utilize a loop cache to relieve the unnecessary pressure placed on the trace cache by loops. Due to the high temporal locality of loops, loops should be cached. We have observed that when loops contain control flow instructions in their bodies it is better to collect traces on a dedicated loop cache instead of using trace cache space. The traces of instructions within loops tend to exhibit predictable patterns that can be detected and exploited at run-time. We propose to capture dynamic traces of loop bodies in a loop cache. The novelty of this loop cache consists of dynamically capturing loop iterations with conditional branches and correlating them to unique loops. Once loop iterations are cached in the loop cache, their bodies can be provided by the loop cache without polluting the trace cache and without any instruction cache accesses. The proposed loop cache includes hardware capable of dynamically unfolding loops such that large traces of instructions are accessed in a single loop cache interrogation.

We evaluate our loop cache and compare it against a baseline machine with a larger first-level instruction cache. We also consider how the loop cache can compliment the introduction of a trace cache by filtering out loop traces that needlessly dominate the trace cache space. We quantify the benefits provided by a fetch engine equipped with the proposed loop cache and unrolling hardware.
In our experiments we explore the design space of a loop cache and associated unfolding hardware and evaluate its efficiency to detect independent iterations in loops in SPECint2000, Media-Bench and MiBench applications. We show that trace cache efficiency and ILP can be significantly improved using our loop caching scheme. This improvement translates into up to 38% performance speedup when compared to a baseline machine with a loop cache and no trace cache to a baseline machine with no loop cache. Further experiments show up to a 16% speedup on a hybrid machine with loop and trace cache compared to a machine with a larger 1 cache and a trace cache.

Resumen

Este trabajo se concentra en el análisis y detección de bucles para incrementar el paralelismo a nivel de instrucciones a través de la especulación de visitas enteras a los bucles. En la tesis se comparan las técnicas propuestas con otras existentes y se proponen técnicas híbridas que explotan las características benéficas de los mecanismos involucrados. Se lleva a cabo un estudio dinámico de las propiedades de muchos conjuntos de aplicaciones con el fin de determinar las características óptimas del hardware propuesto. Tal incluye una memoria cache especialmente diseñada para el almacenamiento y manejo óptimo de instrucciones pertenecientes a los bucles. Proveyendo miles de instrucciones para especulación en la memoria cache de bucles se obtienen aceleraciones en la mayoría de las aplicaciones con el mismo presupuesto de hardware. Se presenta de forma detallada el estudio exhaustivo de técnicas similares así como los detalles del diseño del hardware propuesto. Se justifican cada una de las características basadas en estudios dinámicos de las propiedades de las aplicaciones. También se analizan posibles formas de proveer mayor ganancia en el rendimiento y se presentan alternativas de adaptación del hardware en arquitecturas futuras y en procesadores comerciales existentes.

1 Introduction

To enable wide-issue microarchitectures to obtain high throughput rates, a large window of instructions must be available. A number of techniques have been proposed to support high instruction fetch rates, including compile-time and run-time techniques.

Compile-time techniques produce optimized executables that map more efficiently to the underlying microarchitecture. One common technique employed is called loop unrolling. The popularity of this technique is related to the high instruction-level parallelism that can be extracted from loops. Loops have the property of iterating over the set of instructions located within their bodies, thus high spatial and temporal locality is present in these structures. Compilers can identify and generate copies of loop bodies and then perform software pipelining, producing better instruction schedules and exposing higher levels of instruction level parallelism.

Run-time based techniques apply optimizations while a binary is executing. Hardware techniques can be coupled with compile-time techniques to further reduce execution time. Hardware strategies have the ability to apply optimizations that would not be possible at compile time. Many of these optimizations rely on knowledge of the run-time dynamics of the program. For instance, in loops where the number of loop iterations executed can be deduced at compile time, run-time loop unrolling can be much more effective. Loops that contain ill-structured bodies pose problems for traditional compilation techniques; in such a case, dynamic loop optimization is needed.

From studying a range of general-purpose workloads we have observed that, on average, half of the loops can not be optimized by the compiler. This statistic highlights the potential benefits of capturing loop behavior at runtime and using this information to dynamically unroll loops in hardware.

2 Motivation

Hierarchical memory systems are used to reduce the performance gap between processor speed and memory speed. During the execution of a program, data and instructions are transferred between different levels of the memory hierarchy. When an address is not found in a particular level (i.e., a miss occurs), the next higher level is interrogated for the address. Cache misses can occur for a variety of reasons including first time access, context switches and dynamic control flow changes. In the best case, a cache miss is resolved by the next closest level in the hierarchy. However, in some cases, misses can involve all levels of the hierarchy. Therefore, accesses to data and instructions can have a variable-length latency that is a function on the number of memory levels traversed.

Hardware-based and compiler-based strategies make use of the inherent temporal and spatial locality present in programs to reduce the number of cache misses [1, 2, 3]. In this thesis we study hardware-based loop caching techniques that improve instruction caching efficiency by increasing the fetch bandwidth of the processor front end.

3 Exploiting Loops Locality

Loops are responsible for the majority of the execution time in many classes of applications. Scientific applications can be characterized as spending up to 90% of their execution time in one or a few loops [9]. Compile-time *loop unrolling* [10, 11] is commonly used in scientific programs, and can obtain large speedups. In general-purpose applications (e.g., SPECint programs), many loops are present, but many of these loops have characteristics that make them difficult to unroll at compile time.

By studying the loop characteristics of different workloads, we are able to identify specific features that can expose higher degrees of instruction level parallelism. Some key observations that we have found include that we can accurately predict the number of loop iterations completed across multiple visits to the same loop and the correlation of the pattern of conditional branch outcomes leading up to the loop can serve as an accurate predictor of whether we will enter the loop.

We have also found that there is good correlation between the branches visited during the execution of a single loop iteration and the number of iterations that are completed. Additional correlation patterns exist for nested loop patterns, where the number of iterations performed by the outer loop can be used to predict the number of executions of the associated inner loop.

To perform hardware-based loop unrolling, we must have both the ability to capture past loop behavior, as well as a

way to capture the instructions present in the loop body. For these purposes, we propose to use a hardware-based loop predictor and a hardware-based loop cache. The loop predictor will capture dynamic branch and loop execution such as: the number of iterations executed during a loop visit and the path followed during each iteration.

By capturing this information dynamically, an entire loop's execution can be speculatively loaded into the instruction window. Hence, this mechanism permits building an instruction window containing thousands of instructions. This has the added effect of exposing higher levels of instruction level parallelism

4 Loop Caching Hardware

The hardware to identify a loop body consists of a comparator to detect a negative branch displacement and a stack-based mechanism called the loop stack. At commit time, after loop entry has been detected, the loop's associated loop information is pushed onto the loop stack. As the loop iterates, the loop stack will be updated with *path-in-iteration* information for the loop. This dynamic information includes the *per-iteration in-loop* branch history and the number of completed iterations per visit.

The outcome of every branch executed within a loop iteration is recorded in a *loop prediction table*. At the termination of a loop visit, the loop prediction table holds the complete branch history for all iterations completed during the prior loop visit. Previous work on loop characterization showed that most loops tend to traverse a limited set of paths during a visit, and that these patterns persist over future visits to the loop [15].

Therefore, by recording this set of paths dynamically, we can speculatively generate the entire loop visit, allocating dynamic loop traces in our loop cache. The key observation made is that dynamic loop behavior can be uniquely identified by tracking information for each (*loop head, loop tail*) pair. This characteristic allows us to track the behavior of hot loops in a small, fast, loop cache lookup table.

The information in the loop prediction table is used to unroll [1] the loop in the loop cache. The number of predicted iterations is used as an upper-bound on the number of times that the loop body is unrolled. The collected paths-in-iteration information is used to build traces of instructions to be executed during each iteration.

Figure 1 shows how the loop prediction table is used to capture path-in-iteration information. The path-to-loop register correlates the last n visited branches to the head of a loop. The address of the closest branch outside of the loop body is XOR'ed with the path-to-loop history to index into the loop prediction table.

A tag is maintained in the loop prediction table entry, to prevent any aliasing from occurring in the table (aliasing can lead to mispredictions). A loop is unrolled if the predicted number of loop iterations for the associated visit is larger than 1.

Once dynamic loop behavior is captured, instructions can be allocated in the loop cache. The hardware initiates a prediction by searching the loop prediction table. When a loop is found in the table, we search for a corresponding entry in the loop cache .Figure 2shows the hashing mechanism used to interrogate the loop cache.

The loop table provides the head and the tail addresses for the loop of interest. Then, these fields are XOR'ed to produce a tag that is matched against existing tags in the loop cache lookup table. If a match is found, the loop cache lookup table provides the starting address in the loop cache where the loop has been allocated.

[1] Note that we use the term unrolling here rather liberally, meaning that we can produce multiple iterations through a loop body that may differ upon each iteration.

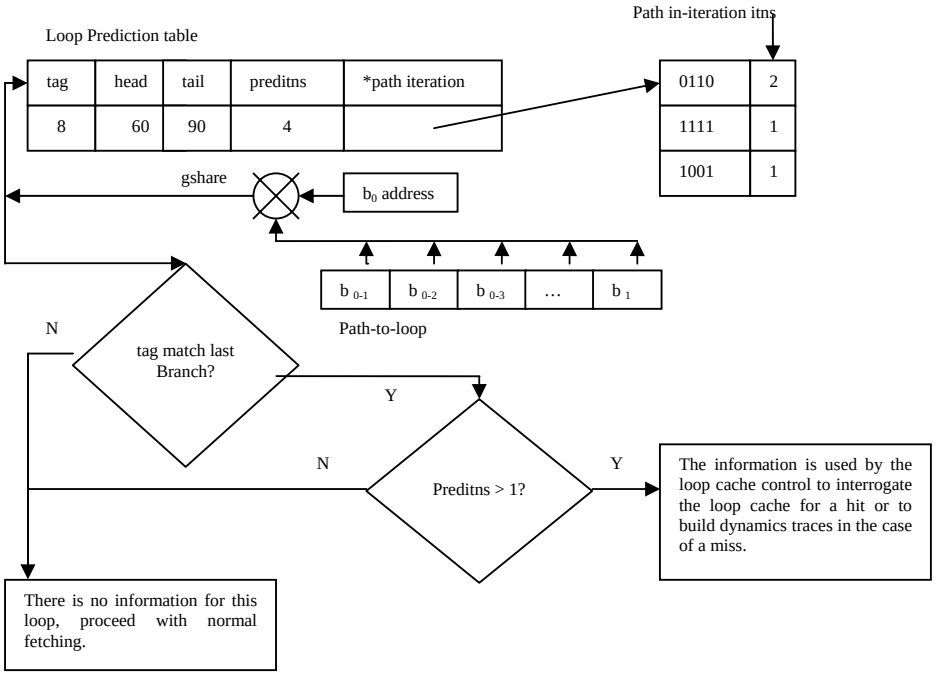

Fig. 1. Indexing the loop prediction table and path-in-iteration table

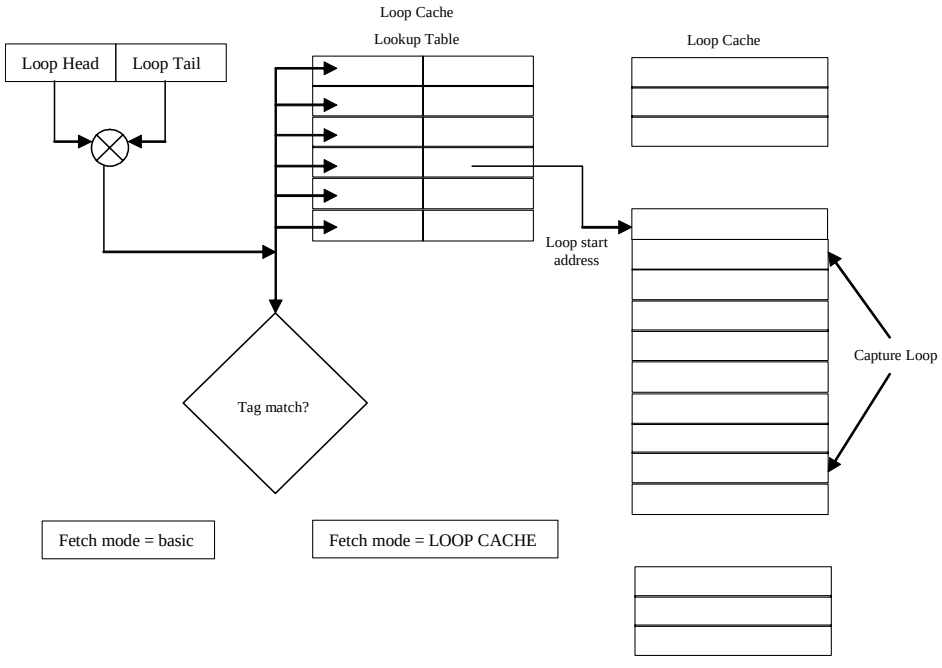

Fig. 2. Interrogating the loop cache. Loop cache access uses a 2-level table structure, first generating a tag to interrogate the loop cache lookup table, which then produces an index into the loop cache Loop cache

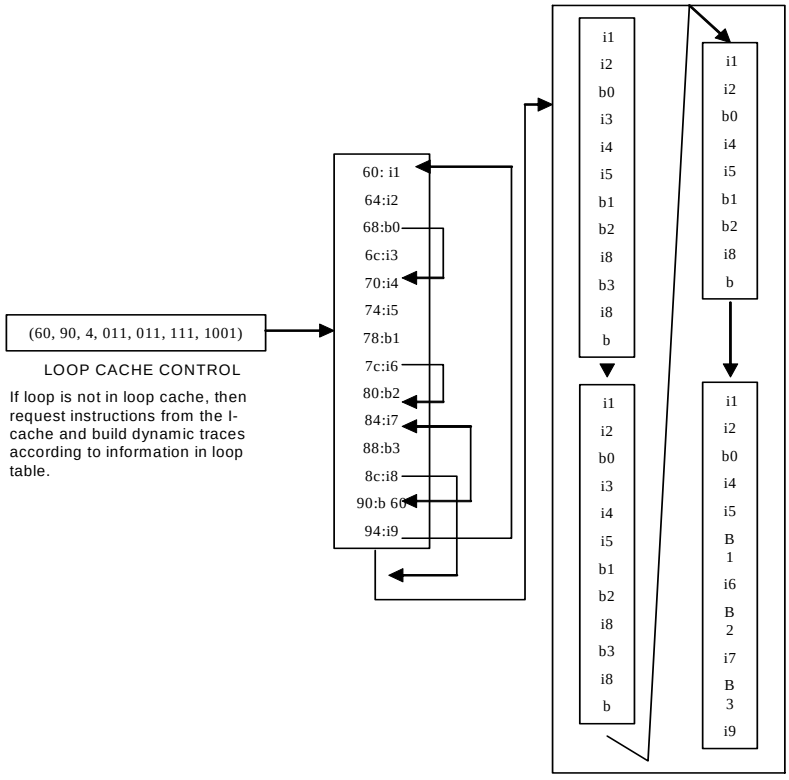

Fig. 3. An unrolled loop visit in the loop cache

When new loops need to be allocated in the loop cache, the loop cache lookup table returns the first unused address. In the case that the loop cache lookup table is full or there is insufficient space available for a loop in the loop cache, LRU replacement is used.

Figure 3 shows an unrolled loop body after having obtained the loopid (head and tail), number of predicted iterations and paths-in-iteration from the loop prediction table (these are included in the loop cache control box). For simplicity, we show the body of the predicted loop as being allocated in the loop cache. The dashed lines indicate the targets of the branches inside the body. The loop cache contains four iterations of this loop. Each box represents a different iteration. The instructions are fetched according to the path-in-iteration information. In this example, our loop body contains four branches. Inspecting the first unrolled iteration of the loop, branches b0, b1, b2 and b3 are predicted as: 0, 1, 1, 0 (0: not taken, 1: taken) respectively. These instructions are shown in the leftmost column at the top of the loop cache. Similarly, the 3subsequent iterations of the loop are unrolled. Instructions are organized in the loop cache according to their dynamic path (very similar to how a trace cache caches traces). Finally, the instructions are sent to the dispatch queue, and from there they are processed by the other pipeline stages.

4.1. Microarchitectural Features for Loop Caching

Our loop caching mechanism was adapted to supply an out-of-order superscalar microprocessor. To evaluate the design space for our loop cache, we developed two models. First, we compare a baseline design that provides only an instruction cache to a system that adds a loop cache. In the second model, our loop cache is integrated with a trace cache, providing a hybrid fetch engine. The purpose of studying this hybrid design is to determine how much a loop cache can benefit a trace cache.

Figure 4 shows the organization of the proposed fetch engine. The fetch mode flag enables one of the three cache entities. Depending on the activated entity, instructions will be sent to the fetch queue through an arbitrated tri-state bus. This bus width is equal to the size of the fetch queue.

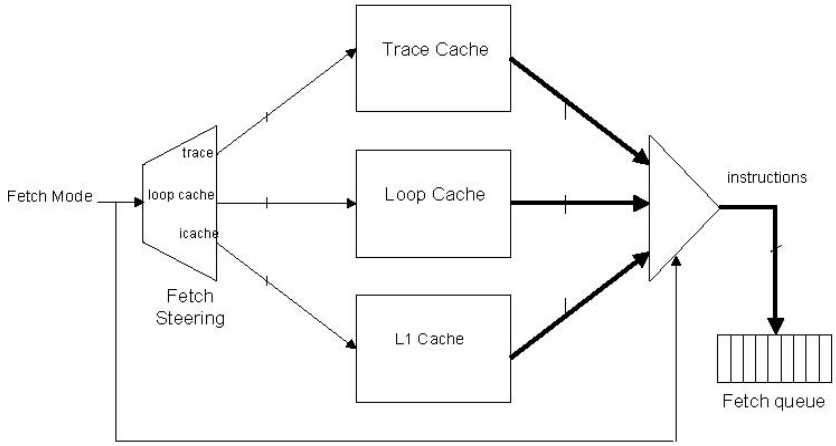

Fig. 4. Multiple source fetch engine

Instruction fetching is initially set to instruction cache fetch mode. In parallel to instruction cache fetching, traces are constructed in the trace cache fill unit. Once traces have been built in the trace cache, the fetch engine interrogates the trace cache and switches to the trace cache fetch mode if a valid trace is found. Similarly, once loops are detected during instruction fetch, history is recorded in the loop cache.

During fetch, the loop cache lookup table is interrogated to detect whether a loop is going to be visited. If a loop entry is detected, then a loop entry flag is set. Later, if at any time the trace cache mispredicts a trace and the loop entry flag is enabled, then the fetch mode is switched to the loop cache. If the flag is not set, then fetch mode is directed to the instruction cache.

Once in loop cache fetch mode, we return to trace cache fetch mode or instruction cache fetch mode on either of the following events: a loop completes a visit, or the loop cache encounters a misprediction. We will switch to trace cache fetch mode as long as we can find a valid trace for the current instruction stream. Otherwise we will return to instruction cache fetch mode

5 Experimental Methodology

The loop caching hardware is implemented for a high performance out-of-order issue superscalar processor with speculative execution. The Simplescalar [16] version 3.0c for the Alpha EV6 architecture was used. The core configuration of the machine utilized for the simulations and the parameters for the loop unrolling hardware are shown in table 1.

Every entry of the loop prediction table occupies 120Bytes (112 bytes to capture the path-in-iteration information of up to 64 iterations per loop visit and up to 8 branches per path-in-iteration, and 8 bytes to hold the tag, loop head and loop tail addresses information).

The proposed issue and fetch rates are equivalent to those proposed in clustered architectures [17]. Every trace cache entry holds 16 instructions, history for 3 branches and address of next instruction when the last instruction in the trace is a taken branch. To hold this information are required 16*4Bytes + 3bits + 32bits per entry. The total space of the trace cache adds up to 133888 Bytes.

PARAMETR	SIZE	PARAMETR	SIZE
Loop Pred. table	4864 B	Loop Stack	1216 B
Path-to-loop length	8 Bits	Pattern Register	Log2 (loop pred. table size)
Loop Cache Size	8 KB	Pathinith Branches	8
Loop Cache lat.	1 cycle	IFQ	16 instrs
Bimod branch pred.	4 K	2lev Gag Branch Pred.	4K
Comb. Branch pred.	4 K	Miss lat.	3 cycles
BTB	1 K	RAS	8 entries
Decode Width	16	Issue Width	16
Commit Width	16	I-Window	256
LSQ	128	Trace Cache	2K enries
L1 Data Cache	16K, 32B, 2-way, LRU	L1 I-cache	16K, 32B, 2-way, LRU
L1Cache lat.	1 Cycle	L2 Data Cache, I-Cache	2M, 64B, 4-way, LRU
L2 Latency	10 cycle	Memory Lat.	250, 2
Memory Bus Width	8 B	Memory ports	4
INT ALU units	16	INT Mult/Divs	8
FP ALU units	8	FP Mult/Divs	4

Table 1. Loop Prediction and Unrolling Hardware Design Parameters

5.1 The Benchmarks

Table 2 includes information of the set of benchmarks used in our evaluation. We have chosen these benchmarks because they contain loops with different properties. Columns contain bench mark suite name, name of application, input dataset, number of dynamic instructions and description of the application.

SUITE	NAME	INPUT	DYNAMIC INSTRS	DESCRIPTION
MiBench	dijkstra	Input.dat	276M	Shosrtest path
MiBench	gsm	Large.au	1000M	Speech transcoding
MiBench	Patricia	Large.udp	477M	A Patricaia Trie
MiBench	fft	Large (8 3278)	245M	FFT computation
MiBench	typeset	Large.out	632M	Typesetting tool
MediaBench	epic	Test. Image.pgm	66M	Image data compression
MediaBench	g271	Clinton.pcm	415M	G.721 voice comprenssion
SPECint2000	vpr	Test	3B	Placement and routing
SPECint2000	gzip	Smred.source	1.66 B	Data compression
SPECint2000	gcc	Test.ccp.i	1.99B	C compiler
SPECint2000	bzip2	Smred.source	2.8B	Data Compression
SPECint2000	parser	Smred.in	214M	Word processor
SPECint2000	twolf	Test	218M	Placement and global

Table 2. Benchmarks from MiBench, MediaBench and SPECint2000

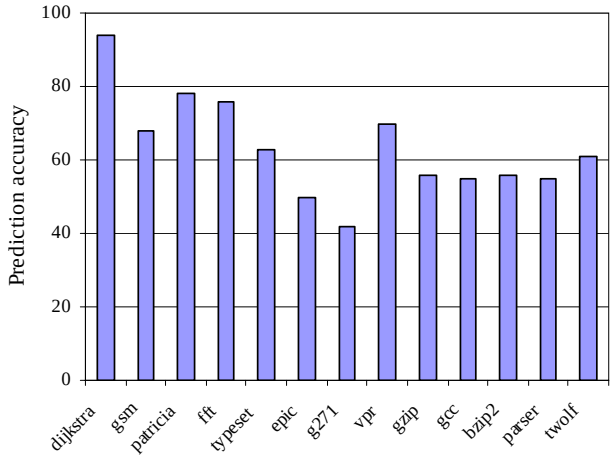

Fig. 5. Prediction accuracy of entering a loop using path-loop correlation

6 Results

6.1 Evaluation of the Loop Prediction Hardware

Before we present performance numbers, we first want to look at the accuracy of the different components described in Section 5. First, we look at how well the hardware can predict a visit to a loop by using path to loop information. Second, we look at how accurately we can predict the number of iterations per visit. Third, we look at how well we can predict paths-in-iteration for each loop iteration. In Figure 5 we look at how effective the path-to-loop pattern can be used to predict a loop visit. Our results are computed on a per loop visit basis. In more than half of the benchmarks we can predict an entry into a loop body 60% of the time, and in 12 out of 13, we predict entry accurately only 50% of the time. At first glance this would seem to be a disappointing result. If the entry branch into a loop is not predictable, the associated loop is not a good candidate for speculation. However, we observed that entry branches that exhibit very high predictability (above 90%) are those that complete the largest number of iterations, and so contribute to a majority of the executed instructions. Thus, the path-to-loop pattern is a very good predictor for identifying hot loop visits (a key for obtaining good performance).

Figure 6 indicates that for most of the benchmarks, we can correctly predict the number of iterations per visit 60%of the time. In some cases we observed that loops that possess unpredictable patterns tend to change their behavior on every loop visit. The results shown here include loops with one or two iterations. These loops are difficult to predict since they do not repeat enough times to warm up the predictor. For speculation purposes, we should avoid caching these loop bodies.

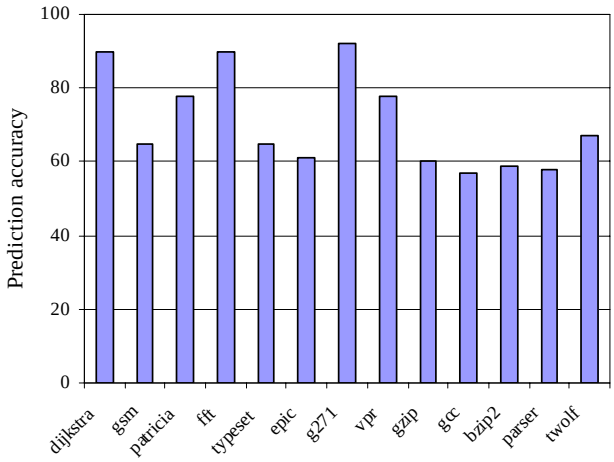

Fig. 6. Prediction Accuracy of Number of Iterations in a Loop Visit. (Results are computed on a per visit basis)

The third characteristic we studied was the predictability of the branches executed during an entire loop visit, which is shown in Figure 7. Even if a single branch is mispredicted, we treat the entire path-in-iteration prediction as wrong. To implement this prediction, a combined predictor was utilized. This predictor uses branch patterns to predict path transitions between neighboring iterations and a two-bit confidence counter to decide whether to change the prediction of individual branches in a path. Next we evaluate the performance benefits of using a loop cache.

6.2 Loop Cache Performance

We compare a machine with a loop cache and an 8KB L1 instruction cache, with a baseline machine with only a 16KB L1 instruction cache. We double the size of the L1 instruction cache in order to compensate for the added space for the loop cache. Figure 8 shows the speedup obtained by the loop cache over the baseline machine. The benchmarks in which larger speedups were obtained are characterized by a smaller number of dynamic loops with predictable patterns [15]. Significant speedups in IPC were obtained in: *fft*, *vpr*, and *bzip2*.Our loop cache machine improves performance across all benchmarks. In the next section we compare the performance obtained for a machine with only a trace cache to a machine with an integrated hybrid fetch engine (loop cache + trace cache).

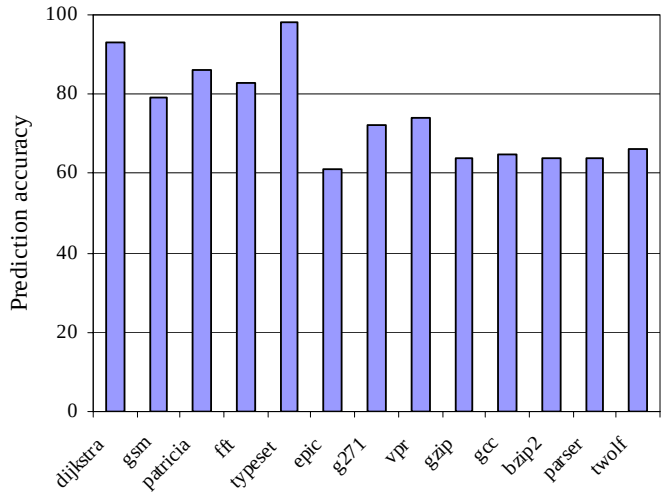

Fig. 7. Prediction Accuracy of path-in-iteration. (Results are computed on a per-iteration basis)

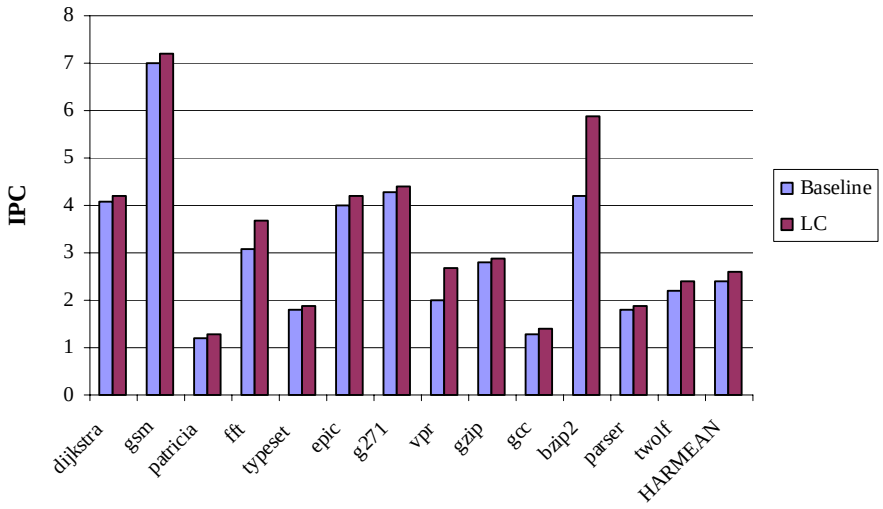

Fig. 8. Comparision of Basline and Loop Cache Machines

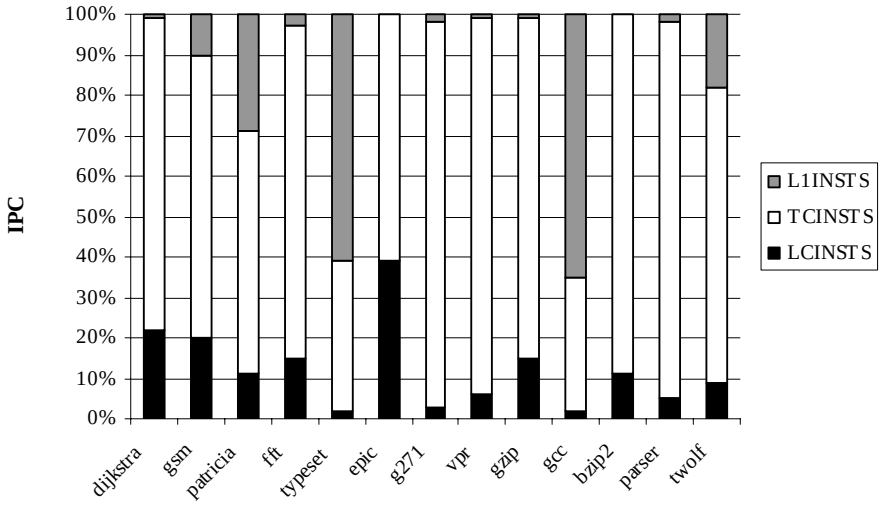

Fig. 9. Distribution frequency of committed instructions

6.3 Performance of a Hybrid Fetch Engine Machine

Next, we want to evaluate how a loop cache cans benefit a trace cache. Besides capturing the IPC for this hybrid configuration, we also want to understand what percentage of the committed instruction stream is provided by a loop cache, and how much is provided by the trace cache. In Figure 9 we show the breakdown of the fetch sources for every committed instruction. As expected, the trace cache captures most of the instructions, since it can also cache instructions outside of loops. The loop cache is capable of caching loops that exhibit a changing behavior; these branches account for approximately 10% of loop executions.

Instructions that are not provided by either of the mechanisms are supplied by the L1 instruction cache. A good example of this is observed in *gcc*, which includes a significant number of indirect branches out of loops.

Lastly, we evaluate an architecture possessing a Hybrid fetch engine and compared it to a machine with a trace cache. The results are presented in Figure 10. We observed significant performance improvements in those benchmarks that have some predictable loops with large iteration counts and with in-loop branches that have more stable behavior (e.g. *gsm* and *bzip2*). For the benchmarks where our mechanism showed only small benefits or slowdown (e.g. *epic* and *gcc*), we found that there were only a limited number of loop bodies with a dynamically changing branch history pattern.

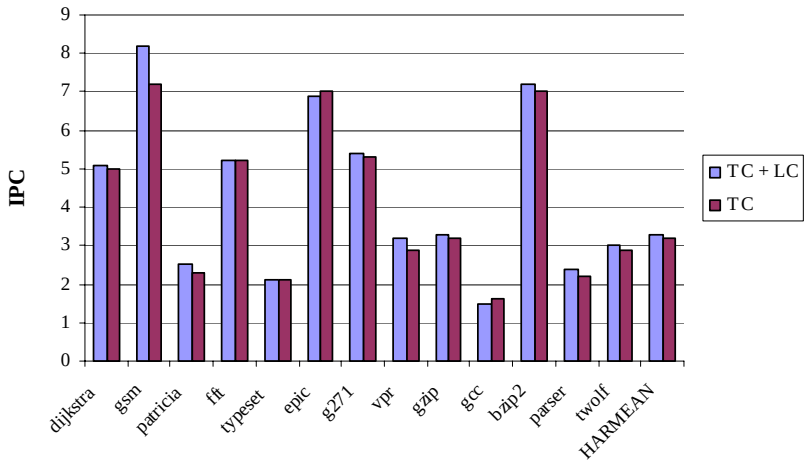

Fig. 10. Comparison of a Trace Cache Machine to a Hybrid Fetch Engine Machine

7 Conclusions

Our loop caching hardware effectively exploits the temporal and spatial locality present in loops. We have shown that a loop cache can signficantly improve instruction level parallelism. We have also seen that a trace cache can obtain more benefit by being coupled to a loop cache system. We developed our prediction strategy based on a thorough study of loop-based behavior.

This has led us to carefully consider the sensitiving of our design to various design parameters. We feel that main contributions of our work are to provide for a simple hardware implementation, while also considering the interaction between other aggressive instruction fetch mechanisms. In future work we will look to smarter steering algorithms that can utilize confidence to suggest when to redirect instruction fetching from either element in our hybrid scheme.

References

1. **K. McKinley, S. Carr, and C.-W. Tseng**, "Improving data locality with loop transformations," ACM Transactions in Programming Languages and Systems, vol. 18, no. 4, pp. 424–453, 1996.
2. **K. S. McKinleyand O. Temam**, "Quantifying loop nest locality using SPEC'95andthe Perfect benchmarks," ACM Transactions on Computer Systems, vol. 17, no. 4, pp. 288–336, 1999.
3. **R. Kessler,** "The alpha 21264 microprocessor," in IEEE Micro, March-April 1999, pp. 24–36.
4. **S.-A. Chi, R.-M. Shiu, J.-C. Chiu, S.-E. Chang, and C.-P. Chung**, "Instruction cache prefetching with extended btb," in IEEE Proc. of Intl. Conf. Parallel and Distributed Systems, December 1997, pp. 360–365.
5. **W.-C. Hsu and J. E. Smith**, "A performance study of instruction cache prefetching methods," in IEEE Transactions on Computers, vol. 47, no. 5, May 1998.
6. **T.-Y. Yeh, D. T. Marr, and Y. N. Patt**, "Increasing the instruction fetch rate via multiple branch prediction and a branch address cache,"in Proc. of the International Conference on Supercomputing, Tokyo, Japan, July 1993, pp. 67–76.
7. **T. M. Conte, K. N. Menezes, P. M. Mills, and B. A. Patel**, "Optimization of instruction fetch mechanisms for high issue rates," in Proc. of the 22nd Annual International Symposium on Computer Architecture, Santa Margherita Ligure, Italy, 1995, pp. 333–344.
8. **R. Rosner, A. Mendelson, and R. Ronen**, "Filtering techniques to improve trace-cache effciency," in Proc. of the International Conference on Parallel Architecture and Compilation Techniques, Barcelona, Spain, September 2001.
9. **J. L. Hennessy and D. A. Patterson**, Computer Architecture: A Quantitative Approach, 2nd ed. Palo Alto, CA: Morgan Kaufmann, 1995.
10. **Aiken and A. Nicolau**, "Loop quantization: An analysis and algorithm," March, 1987.
11. **J. Davidson and S. Jinturkar**, "Improving instruction-level parallelism by loop unrolling and dynamic memory disambiguation,"in Proc. of the 28th Annual International Symposium on Microarchitecture. New York, NY: ACM Press, 1995, pp. 125–132.
12. **K. S. S-T. Pan and J. T. Rahmeh**, "Correlation-based branch prediction," Computer Engineering Research Center, University of Texas at Austin, Tech. Rep. UT-CERC-TR-JTR91-01, August 1991.
13. **S.-T. Pan, K. So, and J. T. Rahmeh**, "Improving the accuracy of dynamic branch prediction using branch correlation," in Proc. of the fifth International Conference on Architectural Support for Programming Languages and Operating System, vol. 27-9. New York, NY: ACM Press, 1992, pp. 76–84.
14. **T. Y. Yeh and Y. N. Patt**, "A comparison of dynamic branch predictors that use two levels of branch history," in Proc. of the 20th Annual International Symposium on Computer Architecture, Goteborg, Sweden, 1993, pp. 257–266.
15. **M. R. de Alba and D. R. Kaeli,** "Runtime predictability of loops," in Proc. ofthe Fourth Annual IEEE International Workshop on Workload Characterization, I. C. Society, Ed., Austin, TX, December 2001, pp. 91–98.
16. **D. Burger, T. M. Austin, and S. Bennett**, "Evaluating future microprocessors: The simplescalar tool set," University of Wisconsin, Madison, Tech. Rep. CS-TR-1996-1308, 1996.
17. **G. J. M. Parcerisa, J. Sahuquillo and J. Duato**, "Effcient interconnects for clustered mi-croarchitectures,"in Proc. of the Int. Conf. on Parallel Architectures and Compilation Techniques (PACT 2002), Charlottesville, Virginia, USA, September 2002, pp. 291–300.

Marcos Rubén de Alba Rosano. *Estudió la carrera de ingeniería electrónica en la Universidad Autónoma Metropolitana Azacpotzalco, completando sus estudios en 1993. Estudió una maestría en Ciencias Computacionales con especialidad en Redes de Computadoras en el Instituto Tecnológico y de Estudios Superiores de Monterrey Campus Estado de México. Continuó sus estudios de posgrado en la escuela Northeastern University en Boston, Massachusetts, Estados Unidos. Ahí completó una segunda maestría en Ciencias Computacionales con especialidad en Sistemas Operativos en el Colegio de Ciencias Computacionales. Realizó estudios de doctorado en ingeniería computacional con especialidad en Arquitectura de Computadoras. Ha sido profesor asociado del Departamento de Electrónica de la UAM Azcapotzalco y profesor de cátedra del ITESM CEM. Impartió cursos a nivel licenciatura en Northeastern University. Tiene experiencia profesional instalando sistemas de automatización y robots en las industrias automotriz y de edificios inteligentes. Asimismo, ha trabajado como investigador en el laboratorio de microprocesadores de Intel en Barcelona, España; donde desarrolló un modelo para cuantifcar y optimizar el consumo de potencia estático de una arquitectura cluster. Sus áreas de investigación involucran el diseño y evaluación de simuladores de microprocesadores a nivel microarquitectura; el desarrollo de sistemas digitales y circuitos integrados VLSI. Realizó sus estudios de posgrado en el extranjero apoyado por Fulbright-CONACYT, la UAM y el FIDERH del Banco de México.*

Dr. David Kaeli. *He received his Ph.D. in Electrical Engineering from Rutgers University in 1992. He received his M.S. in Computer Engineering from Syracuse University in 1985 and a B.S. in Electrical Engineering from Rutgers University in 1981. For the 2001-2002 academic year he spent a year as a Visiting Professor at the Departament of d'Arquitectura de Computadors at the Universitat Politecnica de Catalunya. He is the Director of the Northeastern University Computer Architecture Research Laboratory (NUCAR). He is a Research Thrust Leader for the NSF Center for Subsurface Sensing and Imaging Systems (CenSSIS) . He is also a member of the Northeastern University Institute for Complex Scientific Software . His research looks at the performance and design of high-performance computer systems and software. Current research topics include: profile-guided compilation, high-ILP microarchitectures, database systems, software testing, object-oriented systems design, branch prediction studies, workload characterization, memory hierachy design, embedded systems design, network processors, and software defined radios. He frequently provides tutorials on the subject of Trace-Driven Simulation . He is a member of both IEEE and ACM, serves on the IEEE TCCA Executive Advisory Committee, as ACM SIGMICRO Director of Awards, and as an Associate Editor for both IEEE Transactions on Computers and IEEE Computer Architecture Letters. He is a member of Eta Kappa Nu and Sigma Xi honor societies.*