Available online at www.sciencedirect.com
ScienceDirect
Journal of Systems Architecture 53 (2007) 927–936
JOURNAL OF SYSTEMS ARCHITECTURE
www.elsevier.com/locate/sysarc

Feasibility of decoupling memory management from the execution pipeline

Wentong Li, Mehran Rezaei, Krishna Kavi *, Afrin Naz, Philip Sweany

Department of Computer Science and Engineering, University of North Texas, P.O. Box 311366, Denton, TX 76203, United States

Received 23 August 2006; accepted 6 March 2007
Available online 20 March 2007

Abstract

In conventional architectures, the central processing unit (CPU) spends a significant amount of execution time allocating and de-allocating memory. Efforts to improve memory management functions using custom allocators have led to only small improvements in performance. In this work, we test the feasibility of decoupling memory management functions from the main processing element to a separate memory management hardware. Such memory management hardware can reside on the same die as the CPU, in a memory controller or embedded within a DRAM chip. Using Simplescalar, we simulated our architecture and investigated the execution performance of various benchmarks selected from SPECInt2000, Olden and other memory intensive application suites.

Hardware allocator reduced the execution time of applications by as much as 50%. In fact, the decoupled hardware results in a performance improvement even when we assume that both the hardware and software memory allocators require the same number of cycles. We attribute much of this improved performance to improved cache behavior since decoupling memory management functions reduces cache pollution caused by dynamic memory management software. We anticipate that even higher levels of performance can be achieved by using innovative hardware and software optimizations. We do not show any specific implementation for the memory management hardware. This paper only investigates the potential performance gains that can result from a hardware allocator.

Keywords: Memory management; Software allocators; Hardware allocators; Cache pollution

1. Introduction and motivation

Modern programming languages often permit complex dynamic memory allocation and garbage collection. Such features provide computer systems architects with a challenge of reducing the overheads due to memory management functions. The challenge is further exacerbated by the ever-increasing gap between memory and processor speeds. Some researchers chose to employ custom memory allocation methods into their systems; however, it has been shown that such custom allocators generally do not improve performance [1]. Multithreading has been promoted as a way of tolerating memory latencies. But multithreading has not been used directly to address complex memory management functions.

* Corresponding author. Tel.: +1 940 565 2767; fax: +1 940 565 2799.
E-mail address: kavi@cse.unt.edu (K. Kavi).

1383-7621/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.sysarc.2007.03.003

Even if a dedicated thread (such as a worker thread) were used for memory management, it is unclear whether performance improvement could be achieved, since memory management threads would compete for CPU and cache resources.

Rezaei and Kavi [2] have studied the cache behavior and the pollution of the cache due to the bookkeeping meta-data used by memory allocation functions. This study suggests that if a separate logic is used to perform the memory management functions, the cache performance of the application can be improved. In our research, we advocate the use of separate processing logic to implement memory management functions, whether integrated along with the CPU, incorporated within a memory controller, or embedded within IRAM [3]. We report the impact of a separate hardware allocator on total execution times (not just cache performance). In this paper we do not show either a specific architecture or implementation of the separate hardware memory manager. We only show the potential performance gains resulting from a decoupled hardware memory manager.

We conduct simulations, varying the speed of the hardware allocator. We show that even when the hardware allocator is assumed to take the same number of execution cycles as a software implementation, the decoupled implementation results in performance gains. In some cases, the decoupling produced performance improvements that are larger than the number of CPU cycles consumed if the allocator is implemented in software. We attribute this "super-linear" speedup primarily to the improvement of CPU cache performance, because CPU cache is not polluted by bookkeeping data needed for memory management. However, for some applications, a slow hardware allocator has led to no performance improvements, as the application may have to wait for the allocator. This will not be the case on multithreaded architectures since it would be possible to switch to another active thread while awaiting the allocation. In addition, we will outline some techniques that can be used to eliminate the CPU stalls due to memory management. The elimination of CPU stalls will result in substantial reduction of execution cycles of the application.

The rest of the paper is organized as follows: We present the related research in Section 2; we describe the experimental framework and the benchmarks used in Section 3; we analyze the results in Section 4; we outline some optimization for hardware allocator in Section 5; and we draw our conclusions in Section 6.

2. Related research

Several research threads, including custom allocators, and hardware implementation of memory management function, have influenced our research.

Dynamic memory management is an important problem studied by researchers for the past several decades. Modern programming languages and applications are driving the need for more efficient implementations of memory management functions, in terms of both memory usage and execution performance. Several researchers have proposed and implemented custom allocators and garbage collectors to improve performance of applications requiring dynamic memory management. Berger et al. describe a comprehensive evaluation of custom memory allocators for a wide range of benchmarks including SPECint2000 and memory intensive applications [1]. There are two key findings from their study that are relevant to our research: (1) The total execution time spent on memory management functions can be significant (as high as 41.8% for parser) and (2) Custom allocators do not improve performance when compared to a general-purpose allocator (such as the one by Lea). The first observation is significant since it supports our efforts to decouple memory management functions from the primary execution engine. As for the second finding, while it may be true that software implementations of custom allocators do not seem to improve performance over Lea's allocator, allocators may exhibit different behaviors in terms of cache performance of the allocated objects, and the complexity of hardware implementations. It has been shown that allocators do lead to different cache behaviors of allocated objects [4,5]. Researchers have explored "cache conscious" allocation of objects specifically because of such differences. Hardware implementations of buddy systems (or variances) are easier to implement than more traditional linked-list based allocations. However, cache performance can be poor with such allocators because of the high internal fragmentation inherent with buddy systems. We argue that it is still unanswered as to which allocator provides the best hardware implementation and permits cache conscious memory allocation.

Dynamic storage allocators are traditionally implemented as software within a system's run-time

environment. As previously mentioned, application-specific custom allocators do not necessarily lead to better performance. This suggests that a general-purpose allocator (e.g., Lea allocator) should be explored for hardware implementation. While we do not know of any hardware implementations of Lea, several hardware allocators based on buddy systems have been proposed. Lai et al. [6] and Donahue et al. [7] describe hardware implementations of an Estranged Buddy algorithms, which is a variation of Knuth's Buddy allocator. In Estranged Buddy algorithms, buddies are not immediately combined into larger chunks, thus eliminating the need for later breaking larger chunks into smaller ones. Other hardware implementations of memory management functions have been reported in [8–10]. These studies focus on hardware complexity and the speed of the hardware allocators, but they do not report the actual execution performance gains for applications. Our emphasis is on the performance impact of using hardware allocators for a wide range of benchmarks.

3. Experimental framework

To evaluate the potential for decoupling memory management, we constructed experiments to reflect conditions as close to real execution environments as possible. We have identified and controlled experimental parameters such as machine model(s), appropriate benchmarks, and statistical attributes of interest. In this section we describe our methodology and the selection of benchmarks.

3.1. Simulation methodology

For the purpose of studying the performance implications of decoupling memory management, we extended the SimpleScalar/PISA Tool Set, version 3 [11]. We assumed the existence of hardware for memory management in the form of a separate hardware unit. Our simulated memory management hardware behaves in the same fashion as Lea's allocator [12] used in LINUX system. We further assumed that the hardware functional unit is not pipelined. The later allocation or de-allocation request must wait for the previous requests to finish. In a real implementation, one can use pipelined hardware for memory management to process "bursts" of allocation requests, particularly for applications that allocates several objects together. We added two instructions, "*alloc*" for allocation and "*dealloc*" for de-allocation, to the instruction set of SimpleScalar. The two new instructions are treated the same as other PISA instructions and processed by scheduling them on the separate hardware allocator, viewed as a functional unit (similar to an integer or floating point unit). However, when allocation and de-allocation instructions are encountered, the reservation stations are frozen until the memory management completes and returns the results. This results in CPU stalls, particularly when using a slow hardware allocator. In an actual implementation, this restriction can be eliminated with proper hardware/software optimizations.

Table 1 summarizes our simulation parameters. To explore the feasibility of this decoupling architecture, we used a wide range of allocation latencies (i.e., time to complete an allocation request and return the address of the allocated object), from 1 cycle to 100 cycles, to the number of cycles that match the software allocator.

3.2. Benchmarks

Table 2 shows the benchmark programs used in our experiments. We selected benchmarks for our

Table 1
Simulation parameters

Pipelined CPU parameters	
Issue width	4
Functional units	5 Int (4 ALU, 1 Mult/Div), 5 FP (4 ALU, 1 Mult/Div), 2 Memory, 1 Allocator, 1 Branch
Register update unit size (RUU)	8
Load/store queue size (LSQ)	4
Integer ALU	1 cycle
Integer multiply	4 cycles
Integer divide	20 cycles
FP multiply	4 cycles
FP divide	12 cycles
Branch prediction scheme	Bimodal
Memory parameters	
L1 data cache	4-Way set associative, 16 Kbytes
L1 data cache	4-Way set associative, 16 Kbytes
L1 instruction cache	Direct mapped, 16 Kbytes
L2 unified cache	4-Way set associative, 256 Kbytes
Line size	32 bytes
L1 hit time	1 cycle
L1 miss penalty	6 cycles
Memory latency/delay	18/2 cycles
Allocation time	100 or 1 cycles
De-allocation time	1 cycle

experiments, from SPECInt2000, memory intensive benchmarks and Olden benchmarks. We selected benchmarks exhibiting wide-ranging memory allocation behaviors.

3.2.1. Execution behaviors

The selected benchmarks demonstrate different levels of memory management operations, as a percentage of total execution times (as shown in Table 3). These levels range from very high (parser, cfrac, treeadd) to average (espresso, voronoi), to very low (vortex, gzip, bisort). Table 2 also shows the total number of instructions executed by the benchmarks. Table 3 lists the fraction of execution time spent on memory management functions. Looking at the fraction of time spent on memory management functions, one might assume that this limits the performance gains of a decoupled architecture (i.e., the maximum performance gains using a hardware allocator are limited by the fraction of the time spent on memory allocation functions). However, this is not the case since several complex features of modern architectures impact performance. These factors include cache misses, pipeline stalls, out-of-order execution, and speculations. We will show that these factors may lead to performance gains greater than the fraction of cycles spent on memory management functions.

3.2.2. Nature of memory usage

Wilson showed that applications exhibit different memory allocation and usage patterns [13]. In general, one can classify the patterns as plateau, ramp or random. A plateau application allocates objects at the beginning of its execution and keeps the allocated memory throughout the remainder of the execution. A ramp application allocates objects throughout the course of its execution, but does not de-allocate memory. Thus, the amount of allocated memory increases as program execution progresses. A random application allocates and de-allocates memory throughout its execution. The amount of memory allocated shows a random pattern with peaks and valleys. Benchmarks espresso, and cfrac show random behavior while voronoi and treeadd exhibit plateau behavior. Allocators that take advantage of application behavior can lead to improved performance.

The sizes of objects allocated also impacts the performance of an allocator. Some applications allocate a large number of small objects. For example more than 99% of all objects allocated by cfrac are objects of sizes less than 32 bytes. In some applications, the object sizes vary widely. In espresso, the number of different object sizes exceeds 100, ranging in sizes from 8 bytes to 5 Kbytes, although requests for smaller objects are prevalent. The performance of an allocator implemented in software and as a separate hardware used for memory management depends on the memory allocation, usage behaviors of applications, as well as the sizes of objects allocated. However, investigating the allocators' performance based on allocation behavior of the applications is not the objective of our work; hence, we will not address these issues any further in this paper.

Table 3
Percentage of time spent in memory management

Benchmark name	% Execution time in memory management
164.gzip	0.04
197.parser	20.65
255.vortex	0.59
cfrac	18.75
espresso	11.63
bisort	2.08
treaded	49.44
voronoi	8.75

Table 2
Description of benchmarks

Benchmark family	Benchmark name	Benchmark description	Input	No. of instructions (million)
SPEC	164.gzip	Gnu zip data compression	Test	4540
	197.parser	English parser	Test	1617
	255.vortex	Object oriented database	Test	12,983
MEM	cfrac	Factoring numbers	A 22 digits No.	96
	espresso	PLA optimizer	Mpl4.espresso	73
OLDEN	bisort	Sorting bitonic sequences	250 K integers	607
	treaded	Summing values in a tree	1 M nodes	95
	voronoi	Computing voronoi diagram	20 K points	166

4. Experiment results

In this section, we report the results of our experiments. We discuss both the execution performance and cache behavior resulting from decoupling of memory management functions.

4.1. Execution performance issues

4.1.1. 100-cycle decoupled system performance

We assume that each malloc operation takes fixed 100 cycles in this experiment. Table 4 shows the performance improvements achieved when a separate hardware unit is used for all memory management functions (malloc and free). The second column in the table shows the number of cycles needed on a conventional architecture and the third column shows the execution cycles needed by a decoupled system. The fourth column shows the percentage of speedup achieved by our architecture. The fifth column reproduces the fraction of cycles (from Table 3) spent on memory management functions; we call it percentage of cycles in memory management (CMM for short). The last two columns of the table shows the instructions per cycle (IPC) for hardware and software implementations of memory allocators. In both cases, the IPC does not exceed 1.67. Instruction count of decoupled system is smaller than the conventional architecture, since software implementation of the memory management functions is replaced by the hardware. Smaller IPC in the decoupled system can be due to the non-pipelined implementation of the memory management hardware and the freezing of the reservation stations on malloc requests (see Section 3.1). These restrictions limit the amount of Instruction Level Parallelism (ILP). It means that the number of eliminated cycles is less than the number of eliminated instructions.

Before discussing the range of speedups achieved using a slow allocator (fourth column of Table 4 – 100 cycle Decoupled), we should re-emphasize that 100 cycles for a hardware implementation of memory management functions implies a slow hardware. In contrast, Chang and Gehringer describe hardware that requires, on average, 4.82 cycles [8]. With our slow implementation using 100 cycles, it is possible for the CPU to idle waiting for a memory allocation as reflected by lower IPC counts.

We notice two anomalies when examining the speedup achieved using 100-cycle hardware implementation (fourth column of Table 4). First, for some benchmark programs (vortex, bisort), even a 100-cycle hardware memory manager achieves higher performance than the fraction of cycles spent on memory management functions by a software implementation (comparing columns 4 and 5 of Table 4). We will show shortly that this is in part due to the CPU cache misses eliminated by moving memory management to dedicated hardware. The second anomaly is for voronoi and treeadd programs; the decoupled system shows performance degradation for these benchmarks. We believe that this is due to two factors: (1) CPU stalls resulting from a slow allocator (compare the IPCs) and (2) the allocation behavior of the application. The software allocator (Lea's) takes advantage of the allocation behavior of these programs. These programs perform all of their allocations at the beginning of the execution and keep all the allocated memory throughout the execution. In addition, most allocated objects are small and belong to 8, 16 or 32 byte chunks. These sizes can be allocated very fast in Lea's allocator.

4.1.2. 1-cycle decoupled system performance

Table 5 shows the execution speedup achieved assuming 1-cycle for all memory management

Table 4
Execution performance of separate hardware for memory management

Benchmark name	CC (cycle count) million		Speedup (%)	CMM (%)	IPC (instructions per cycle)	
	CONV	Decoupled			CONV	Decoupled
164.gzip	2725	2724	0.0309	0.04	1.67	1.67
197.parser	1322	1280	3.19	20.65	1.57	1.26
255.vortex	12,771	12,602	1.34	0.59	1.00	1.03
cfrac	107	99	7.83	18.75	1.16	0.96
espresso	46	45	1.44	11.63	1.18	1.46
bisort	474	426	10.03	2.08	1.31	1.42
treaded	134	165	−23.19	49.44	1.59	0.58
voronoi	123	123	−0.01	8.75	1.38	1.23

Table 5
Execution limits of separate hardware for memory management

Benchmark name	CC (cycle count) million		Speedup (%)	CMM (%)	IPC (instrucitons per cycle)	
	CONV	Decoupled			CONV	Decoupled
164.gzip	2725	2724	0.03	0.04	1.67	1.67
197.parser	1322	1074	18.81	20.65	1.57	1.51
255.vortex	12,771	12,591	1.41	0.59	1.00	1.03
cfrac	107	78	27.65	18.75	1.16	1.23
espresso	46	39	14.85	11.63	1.18	1.67
bisort	474	413	12.76	2.08	1.31	1.47
treaded	134	63	52.65	49.44	1.59	1.51
voronoi	123	111	10.37	8.75	1.38	1.37

functions. This data places an upper bound on performance improvement for decoupled memory management architecture. We will discuss some techniques for achieving faster allocators later in this paper. Such implementations would eliminate CPU stalls awaiting an allocation, since allocations take only one cycle.

Note that eliminating the CPU stalls using a 1-cycle hardware implementation produces a "super-linear" speedup for almost all the benchmarks (fourth column, compared with fifth column of Table 5). The speedup for the 1-cycle decoupled system should be at least the same as the percentage of cycles spent in memory management (CMM) in convectional architecture. This can be viewed as linear speedup. If the percentage of speedup greater than the percentage of the CMM, the system has achieved a super-linear speedup. According to the data shown in Table 5, 1-cycles decoupled system reveals super-linear speedup for all the applications except gzip and parser. We attribute the super-linear performance to the removal of conflict (cache) misses between the memory allocation functions and the applications. In Section 4.2, we have also investigate the first level cache performance of our selected benchmarks.

4.1.3. Lea-cycle decoupled system performance

Table 6 shows the average number of cycles spent per malloc call when a software implementation of the Lea allocator is used. Note that the second column of Table 6 shows the average number of CPU cycles per memory management function (not the percentage shown in the other tables thus far). In our experiments thus far we have used a fixed number of cycles (either 100 or 1) for each allocation when implemented in hardware. However, as shown in Table 6, software allocators take different amounts of times for allocation, depending on the size of the object and the amount of search needed to locate a chunk of memory sufficient to satisfy the request. We repeated our experiments using the same average number cycles for a hardware allocator as that for the software implementations respectively (second column of Table 6). The performance gains of these experiments are shown in the third column of Table 6.

Table 6
Average number of malloc cycles needed by Lea allocator

Benchmark name	Average CMM	% of Speedup
164.gzip	790	0
197.parser	69	10.02
255.vortex	401	0.88
cfrac	93	8.8
espresso	87	4.08
bisort	90	10.95
treaded	67	0
voronoi	79	2.22

Based on the data shown in Table 6, we classify the benchmarks into three groups. The first group consists of benchmark with an average number of cycles per memory management request exceeding 100 cycles (viz., gzip and vortex). For these types of benchmarks, the performance of Lea's allocator is poor since they allocate objects with very large sizes. Lea's allocator has to request memory from the system for each large object. The second group, which includes the majority of the benchmarks, requires less than 100 cycles per memory management request, and this group includes parser, cfrac, espresso, bisort, and voronoi. For these applications, even when the number of cycles needed per memory allocation by the hardware allocator is set equal to those of a software allocator, the performance gained by the decoupled allocator is noticeable. The third group of applications that includes treeadd, generate allocation requests in burst (sev-

eral allocation requests in sequence). For these applications, our current hardware allocator causes CPU stalls since our hardware is not pipelined, resulting in performance degradations.

4.2. Cache performance issues

Previously we stated that the "super-linear" speedup with separate 100-cycle hardware for memory management functions (at least for vortex and bisort) is due in part to the elimination of CPU cache misses. We now explore this in more detail. Tables 7 and 8 show the data for L-1 instruction and data caches.

The reduction in instruction cache misses can be more easily understood since instructions comprising malloc and free functions implemented in software are removed from the execution pipeline. The reduction in data references and misses (Table 8) is because the allocation bookkeeping meta-data maintained by the allocator is no longer brought into CPU cache. Our results are similar in spirit to those of [2], but differ in actual values.

Using miss penalties from Simplescalar, as well as the memory accesses eliminated (both from Instruction and Data caches), we can estimate the number of cycles eliminated from CPU execution. This should indicate the performance contribution due to improved CPU cache performance. For example, for vortex, the elimination of some memory accesses for instructions and data as well as the reduction in cache misses has contributed to 2% of the 2.81% improvement shown in Table 4; the remaining performance is mostly due to the elimination of instructions from the execution pipeline. Note that for vortex, since this application shows a CPI close to 1 cycle on average, computing the contribution of reduced cache misses to the overall performance gains is straightforward. Similar computations can be used to find the performance gains due to improve cache performance for other benchmarks; however such computations are more complex because an IPCs that is not equal to one reflect out-of-order execution of instructions.

5. Simple optimization of the decoupled memory manager

In general, a hardware implementation of any function should require fewer cycles than a corresponding software implementation. The performance of a hardware implementation of Lea's allocator can also be improved for applications such

Table 7
L-1 instruction cache behavior

Benchmark name	Conventional architecture		Decoupled architecture	
	No. of references (million)	No. of misses (thousand)	No. of references (million)	No. of misses (thousand)
164.gzip	5145	70,412	5144	70,356
197.parser	2320	10,841	1825	6040
255.vortex	14,148	974,678	14,094	959,584
cfrac	140	7048	107	4122
espresso	86	1286	77	779
bisort	697	1.1	700	1.08
treaded	257	1.3	124	0.98
voronoi	187	1023	174	1214

Table 8
L-1 data cache behavior

Benchmark name	Conventional architecture		Decoupled architecture	
	No. of references (million)	No. of misses (thousand)	No. of references (million)	No. of misses (thousand)
164.gzip	1504	37,616	1504	37,577
197.parser	927	11,659	677	8298
255.vortex	6920	70,412	6875	68,828
cfrac	50	10	37	9.9
espresso	23	94	20	74
bisort	161	2193	156	2193
treaded	88	1086	40	1056
voronoi	58	1054	33	928

Table 9
Performance due to predicted pre-allocation

Benchmark name	Percentage of speedup (%)	
	1 cycles	100–10 cycles
164.gzip	0.035	0.031
197.parser	18.80	14.53
255.vortex	2.90	1.37
cfrac	27.64	11.71
espresso	14.06	6.40
bisort	12.76	12.57
treaded	52.67	49.22
voronoi	10.37	10.36

as voronoi and treeadd which make bursts of malloc calls. In such cases the (hardware) allocator could predict that the next malloc request would be for the same sized object as the previous request. Thus the allocator could pre-allocate similar-sized objects. If the prediction were correct, future malloc requests would be satisfied very quickly, say in 10 cycles, instead of 100 cycles. If the prediction were incorrect, the next malloc would consume 100 cycles. The last column of Table 9 shows the results using this predicted allocation technique.

The behavior observed is similar to Balakrishnan and Sohi [14] that states that for some SPEC CPU2000 benchmarks, 95% of the calls to malloc are for the same-sized of objects as other malloc calls. The data clearly shows that prediction works well for treeadd and vernoi, that have shown degraded performance when using an 100-cycle hardware. In most cases the results are very close to those of a 1-cycle hardware implementation data (compare second and third columns of Table 9), suggesting that it is possible to achieve faster hardware performance for memory management even without assuming very fast hardware. Reducing the average latency of memory allocation requests will also reduce CPU stalls. For cfrac and espresso, prediction is not very accurate because of the memory usage patterns. However, we get about 25% performance improvements over the case where the hardware allocator was assumed to take the same number of cycles as its software counterpart.

6. Conclusions and future research

In this study we have shown that decoupling memory management functions from the processing pipeline can lead to improved performance. Several features impact performance of modern architectures. Among these are out-of-order execution, speculative execution, and cache hierarchies. Application characteristics in terms of memory usage, distribution of allocation requests over the lifetime of the application, and the sizes of objects requested also impact performance. Decoupling eliminates a fraction of the instructions and data accesses from the primary processing element, thus improving cache performance. The processing pipeline may freeze if malloc requests come in bursts. We explored the impact of these issues and presented data to reflect the contributions due to various factors.

In this study, we only explored the performance gains possible by decoupling memory management functions from the processor pipeline. We have not explored the various implementation alternatives of memory management in hardware. The memory processor can be embedded in a DRAM or included in a memory controller. The memory processor may even be integrated on the same die as a CPU. The implementation can be based on a general-purpose pipelined processing unit or an ASIC. If a general-purpose processor is used, it becomes possible to support user-level custom allocators in decoupled memory management architecture.

Our study paves the way for many interesting avenues of further research to fully benefit from a decoupled architecture. Performance of hardware allocators can further be improved with additional techniques. As shown in this paper, for some applications the CPU stalls on mallocs. Techniques to avoid such stalls are needed. The malloc requests may be scheduled well ahead of their actual need, thus overlapping the delay of actual allocation. This is somewhat similar to the program demultiplexing [14], where malloc calls were speculatively executed using threads. Such techniques will not lead to significant performance improvements in conventional architectures (without a separate processing engine dedicated to memory allocations), since instructions comprising malloc functions will be executed utilizing the processing resources. Multithreaded architectures in general and SMT in particular, can benefit from a decoupled allocator; SMT can issue instructions from other threads while waiting malloc for a thread [15].

Innovative memory management algorithms aimed specifically for hardware implementation is another potential avenue of research. Issues related to dynamic frequency and voltage control to manage energy consumed by the primary execution unit and the allocator can yield energy savings for gen-

eral-purpose applications. In addition to finding faster implementations or using the prediction technique shown in this paper, it may also be possible to send stores to memory processor even before an object allocation is complete. Thus stores can be committed without delays. This is somewhat similar to Load Squared architecture proposed by Hewlett Packard [16]. It may also be possible to find new cache organizations. The memory processor can track memory usage and improve cache performance of the allocated data.

Thus, we believe that a decoupled architecture as proposed in this research can be the building block for the next-generation general-purpose processors.

References

[1] E.D. Berger, B.G. Zorn, K.S. McKinley, Reconsidering custom memory allocation, in: Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications, Seattle, USA, 2002, pp. 1–12.

[2] M. Rezaei, K.M. Kavi, Intelligent memory management eliminates cache pollution due to memory management functions, Journal of Systems Architecture 52 (1) (2006) 41–55.

[3] D. Patterson et al., The case for intelligent RAM: IRAM, IEEE Micro (April) (1997) 34–44.

[4] K.M. Kavi, M. Rezaei, R. Cytron, An efficient memory management technique that improves localities, in: Proceedings of the 8th International Conference on Advanced Computing and Communications (ADCOM 2000), Cochin, India, December 14–16, 2000, pp. 87–94.

[5] Y. Feng, E.D. Berger, A locality-improving dynamic memory allocator, in: Proceedings of the 2005 workshop on Memory System Performance (MSP 2005), Chicago, USA, 2005, pp. 68–77.

[6] V.H. Lai, S.M. Donahue, R.K. Cytron, Hardware optimizations for storage allocation in real-time systems, Tech Rept, #77, Department of Computer Science and Engineering, Washington University, St. Louis, Mo, 2003.

[7] S. Donahue, M. Hanpton, R. Cytron, M. Franklin, K. Kavi, Hardware support for fast and bounded-time storage allocation, in: Proceedings of the Second Workshop on Memory Performance Issues (WMPI 2002), Anchorage, USA, 2002.

[8] J.M. Chang, E.F. Gehringer, A high-performance memory allocator for object-oriented systems, IEEE Transactions on Computers 45 (3) (1996) 357–366.

[9] H. Cam et al., A high performance hardware efficient memory allocator technique and design, in: Proceedings of the International Conference on Computer Design, Austin, USA, 1999, pp. 274–276.

[10] M. Shalan, V. Mooney, A dynamic memory management unit for embedded realtime system on a chip, in: Proceedings of the International Conference on Compilers, Architecture and Systhesis for Embedded Systems (CASES), November 2000, pp. 180–186.

[11] D. Burger, T.M. Austin, The SimpleScalar Tool Set, Version 2.0, Tech. Rep. CS-1342, University of Wisconsin-Madison, June 1997.

[12] D. Lea, A Memory Allocator, <http://gee.cs.oswego.edu/dl/html/malloc.html>.

[13] P.R. Wilson, M.S. Johnstone, M. Neely, D. Boles, Dynamic storage allocation: a survey and critical review, Lecture Notes in Computer Science 986 (1995).

[14] S. Balakrishnan, G. Sohi, Program Demultiplexing: Data-flow based speculative parallelization of methods in sequential programs, in: Proceedings of the 33rd International Symposium on Computer Architecture, June 2006, pp. 302–313.

[15] D. Tullsen, et al., Simultaneous multithreading: maximizing on-chip parallelism, in: Annual International Symposium on Computer Architecture (ISCA-22), June 1995, pp. 392–403.

[16] S. Yehiay, J. Collardz, O. Temamy, Load square: adding logic close to memory to reduce the latency of indirect loads with high miss ratio, MEDEA Workshop, held in conjunction with PACT-2004.

Wentong Li received his MS degree in Computer Science from the University of North Texas. He is currently completing his PhD in Computer Science at the University of North Texas. He is employed by Turn, Inc. as staff software engineer. His research interests cover the areas of computer architecture, machine learning and information retrieval.

Mehran Rezai received BS and MS degrees in Electrical Engineering from the University of Alabama in Huntsville and a PhD in Computer Science from the University of North Texas. He worked as a visiting faculty member at the University of Texas at Arlington. He is currently working as a software consultant in Washington, DC area.

Krishna M. Kavi is currently a professor and the Chair of Computer Science and Engineering Department at the University of North Texas. Previously, he was the Eminent Scholar Chair Professor of Computer Engineering at the University of Alabama in Huntsville from 1997 to 2001. He was on the faculty at the University of Texas at Arlington from 1982 to1997. He was a program manager at the US National Science Foundation from 1993 to 1995. He has extensive research record covering intelligent memory systems, multithreaded and decoupled archi-

tectures, dataflow model of computation, scheduling and load-balancing.

Afrin Naz is currently completing her PhD in Computer Science at the University of North Texas. She received her MS in Computer Science from Midwestern State University, Wichita Falls, Texas. She is a member of *UPSILON PI EPSILON Chapter of Texas* at Midwestern State University. She is also the recipient of multi-cultural scholastic award of University of North Texas. She will start her academic career as an Assistant Professor at Drake University in Iowa, in Fall 2007. Her research interest includes Computer Architecture, Compilers and Embedded System designs.

Philip Sweany, associate professor in UNTs Computer Science and Engineering Department, maintains a research focus in both compiler optimization for architectures exhibiting fine-grained parallelism and application of compiler optimization algorithms to automated synthesis of net-centric systems.