Cache Impacts of Datatype Acceleration

Lisa Wu, Martha A. Kim, Stephen A. Edwards
Department of Computer Science, Columbia University, New York, NY
{lisa,martha,sedwards}@cs.columbia.edu

Abstract—Hardware acceleration is a widely accepted solution for performance and energy efficient computation because it removes unnecessary hardware for general computation while delivering exceptional performance via specialized control paths and execution units. The spectrum of accelerators available today ranges from coarse-grain off-load engines such as GPUs to fine-grain instruction set extensions such as SSE. This research explores the benefits and challenges of managing memory at the data-structure level and exposing those operations directly to the ISA. We call these instructions *Abstract Datatype Instructions* (ADIs). This paper quantifies the performance and energy impact of ADIs on the instruction and data cache hierarchies. For instruction fetch, our measurements indicate that ADIs can result in 21–48% and 16–27% reductions in instruction fetch time and energy respectively. For data delivery, we observe a 22–40% reduction in total data read/write time and 9–30% in total data read/write energy.

Index Terms—Accelerators, Instruction Set Extensions, Data Structures, Cache Hierarchy

1 Introduction

In response to strict power constraints and diminishing single-core performance returns, the hardware industry has shifted *en masse* to parallel multicore chips. In theory, parallel processing can match historic performance gains while meeting modern power budgets, but as recent studies show, this requires near-perfect application parallelization [1]. In practice, such parallelization is often unachievable: most algorithms have inherently serial portions and require synchronization in their parallel portions. Furthermore, parallel software requires drastic changes to how software is written, tested, and debugged.

Application-specific integrated circuits (ASICs) are the gold standard for computational power and performance efficiency, but it is uneconomical and impractical to produce a custom chip for every application. As a result most chips are likely to remain programmable, and, ideally, deliver custom-caliber efficiency in a general-purpose setting. Hardware accelerators, such as graphics coprocessors, cryptographic accelerators [2], or network processors [3], [4], provide this for their target domain, but often have awkward, ad hoc interfaces that make them difficult to use and impede software portability.

We explore the impact of supplementing general-purpose processors with *abstract datatype processors* (ADPs) to deliver custom hardware performance in a form palatable to software. ADPs implement abstract datatype instructions (ADIs) that expose to hardware high-level types such as hash tables, XML DOMs, relational database tables, and others common to software.

By encapsulating data and algorithms richer than the usual fine-grained arithmetic, memory, and control-transfer instructions, ADIs provide ample implementation optimization opportunities in the form of an already familiar programming interface. Architects have made heroic efforts to quickly execute streams of fine-grained instructions, but their hands have been tied by the narrow scope of program information that conventional ISAs afford to hardware. ADIs release these restraints. Good software programming practice has long encouraged the use of carefully written, well-optimized libraries over manual implementations of everything; ADIs simply supply such libraries in a new form.

Manuscript submitted: 17-Sep-2011. Manuscript accepted: 14-Oct-2011. Final manuscript received: 21-Oct-2011

2 Motivation

It remains an open question how best to provide high-performance, energy-efficient, single threaded computation. We highlight the following three lessons, which motivate this new direction of research.

- *Dark silicon projections.* Power and cooling are projected to allow only a small fraction of transistors to be fully operational at any time [5], [6]. With power more valuable than area, systems incorporating specialized hardware begin to make far more sense.
- *Serial performance still matters.* Recent work has shown parallel speedups require near-perfect application parallelization [1]; high-performance, low-power, sequential execution remains important.
- *Do not neglect programmability.* The multicore revolution was driven by hardware needs, imposing difficult changes on software. Yet proper software technology can mitigate the problems that might otherwise hinder the adoption of promising new architectures: compilers for MIPS and other early RISC ISAs shielded programmers from the attendant increases in instruction count.

Together these three lessons inspire our application of abstract datatypes to bridge the divide between applications and specialized hardware resources.

Posted to the IEEE & CSDL on 11/10/2011
DOI 10.1109/L-CA.2011.25 1556-6056/11/$26.00 © 2011 Published by the IEEE Computer Society

TABLE 1
Example Abstract Datatype Instructions for Hash Tables

ADI	Description
`new` *id*	Create a table; return its ID in register *id*
`put` *id*, *key*, *val*	Associate *val* with *key* in table *id*
`get` *val*, *id*, *key*	Return value *val* associated with *key* in table *id*
`remove` *id*	Delete hash table with the given ID

3 Architecture and Design

ADIs are instructions that express hardware-accelerated operations on data structures. Table 1 shows example ADIs for a hash table accelerator. The scope and behavior of a typical ADI resembles that of a method in an object-oriented setting: ADIs create, query, modify, and destroy complex data types, operations that might otherwise be coded in 10s or 100s of conventional instructions. Multiple studies in a range of domains conclude that the quality of interaction across an application's data structures is a significant determinant of performance [7], [8], [9]. In this work we consider ADIs for sparse vector and hash table types.

As with other instruction set extensions, we assume a compiler will generate binaries that include ADIs where appropriate. When an ADI-enhanced processor encounters an ADI, the instruction and its operand values are sent to the appropriate ADP for execution. For example, operations on hash tables would be dispatched to the hash table ADP; operations on priority queues would be dispatched to the priority queue ADP. While ADIs can be executed in either a parallel or serial environment, we only consider single-threaded execution here.

4 Evaluation Methodology

In this paper, we quantify the impact of ADIs on instruction and data delivery to the processing core via the memory hierarchy. Storage arrays, caches, and the data movement associated with these structures consume a substantial fraction of total computation energy. In one RISC processor, each arithmetic operation consumed 10 pJ, but reading the two operands from the data cache required 107 pJ each and writing the result back required 121 pJ [10]. Thus, 335 pJ is spent on the L1 data cache to accomplish a 10 pJ arithmetic operation. Amdahl's law dictates that we should optimize these operations to the extent possible.

We examine two contemporary, performance-critical, serial applications that are not obviously amenable to parallelization: support vector machines and natural language parsing.

- Machine learning classification is used in domains ranging from spam filtering to cancer diagnosis. We use LIBSVM [11], a popular support vector machine library that forms the core of many classification, recognition, and recommendation engines. In particular, we used LIBSVM to train a SVM for multi-label scene classification [12]. The training data set consists of 1211 photographs of outdoor scenes belonging to six potentially overlapping classes, *beach*, *sunset*, *field*, *fall foliage*, *mountain* or *urban*. We target the sparse vector type with an ADP with specialized instructions for insertion, deletion, and dot product operations on sparse vectors.
- Parsing is a notoriously serial bottleneck in natural language processing applications. For this research, we selected an open source statistical parser developed by Michael Collins [13]. We trained the parser using annotated English text from the Penn Treebank Project [14] and parsed a selection of sentences from the Wall Street Journal. For this application we target hash tables, assuming ADI support for operations such as table lookup and insertion.

We instrumented these two applications using PIN, and fed the dynamic instruction and data reference streams to a memory system simulator. We then combined the output access counts with Cacti's characterization of the access time and energy of each structure to compute the total time and energy spent fetching instructions and data.

We evaluate a design space of fifty-four cache configurations for ADI-enhanced and ADI-free instruction streams. We considered direct-mapped and 2-way L1 caches of capacity 2 KB to 512 KB (each with 32 B lines), and unified L2 caches (each with 64 B lines) of sizes 1 MB (4-way), 2 MB (8-way), and 4 MB (8-way). We fixed main memory at 1 GB.

5 Instruction Delivery

First, we compare the instruction fetch behavior of an ADI-equipped processor to its ADI-free counterpart. We characterize the hierarchy by total energy consumed, dynamic and leakage, over all levels of the hierarchy; and total time spent accessing the memory system.

Figure 1 shows the results of our instruction fetch experiments. Graphs on the left represent the SVM application; those on the right are for the Parser benchmark. The scatter plots in the first row graph the total instruction fetch energy against the total instruction fetch time. Here, the crosses show the instruction cache behavior for ADI-free programs; the circles show the change in efficiency with the addition of ADIs.

Of the two programs, SVM shows the greatest improvement, confirming the importance of the sparse vector dot product in the execution of this benchmark. The Parser benchmark shows more modest improvements, reflecting the smaller fractional importance of the hash table datatype in this application. From this design space we identify the set of Pareto-optimal cache designs: three from SVM and four for Parser. Selecting the optimal cache configurations *for each benchmark* is optimistic, as in reality many applications will share a single configuration; we do so here in order to measure ADIs against the *best possible performance a cache can offer*.

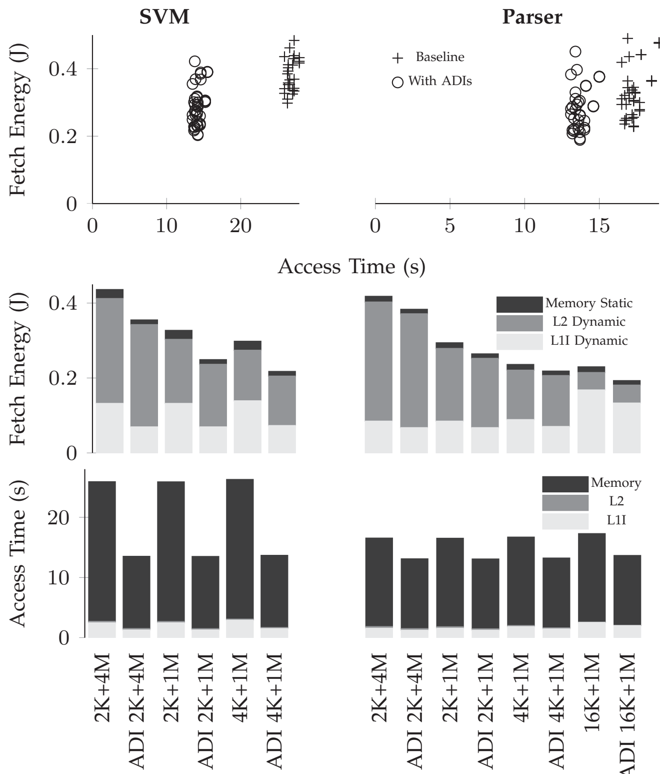

Fig. 1. ADIs' impact on instruction fetch energy and time.

The remaining charts in Figure 1 show the detailed breakdown of energy and instruction fetch time of each optimal cache configuration. The energy consumption is divided into static and dynamic components for each level of the hierarchy. When a program's instruction footprint does not fit into the L1 instruction cache (L1I), we see L2 dynamic energy dominate. However, the dynamic L1I energy becomes dominant as the L1I size increases and the working set begins to fit.[1]

The bottom row of Figure 1 shows the overall time the memory system spends delivering instructions. Because main memory is far slower than the L1 cache, even the minuscule number of instruction fetches that go to main memory after missing both caches tend to dominate.

To summarize our findings on instruction fetch, each application gains in performance over all Pareto optimal cache designs, with the improvements in performance and energy savings coming in proportion to the reduction in total instructions fetched. To the extend instruction fetch consumes time and energy, these results suggest that changing the instruction encoding can reap important benefits, regardless of application domain.

6 Data Delivery

Because ADIs encapsulate data structures as well as the algorithms that act on them, they can be implemented

1. We omitted bars for main memory dynamic energy and L1I and L2 cache static energy as they were roughly a thousand times smaller—not visible on these graphs.

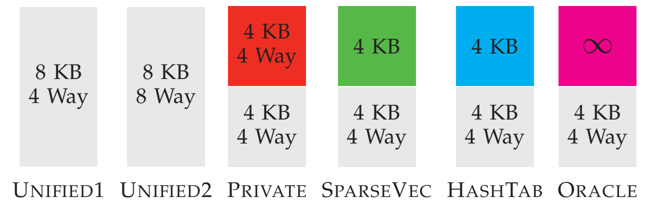

Fig. 2. Compared configurations of level 1 data stores

using specialized datapaths coupled to special-purpose storage structures that can be considerably more efficient than the general-purpose alternative. While there has been a great deal of research on specialized datapaths and computation, few researchers have considered specializing the memory system. Below, we examine the costs and benefits of segregating and serving streams of data according to its type.

We compare several different L1 data cache organizations while keeping the other levels of the hierarchy fixed. We hold the total L1 resources (i.e., total number of bits) constant but deploy them in several ways, illustrated in Figure 2: as a single, unified L1 cache (UNIFIED1 and UNIFIED2); two identical, private caches (PRIVATE); and one normal cache plus one type-specific storage unit (SPARSEVEC and HASHTAB, described below). To provide an upper bound on the data access savings one can hope to see, we also model an infinite, instantaneous, zero-energy storage unit to serve type-related memory requests (ORACLE).

We consider a naïve implementation of a sparse vector store, which consists of a RAM storage array and a small number of registers that hold pointers to the next element in a particular sparse vector. The processing core can issue two types of requests to the sparse vector store: *begin vector*, which notifies the vector store that the processor is about to initiate an operation of the vector at a particular base address; and *next element* events, through which the processor requests the next element (i.e., index, value pair) in a sparse vector.

Our experimental results indicate there is significant benefit both in data delivery speed and energy consumption from issuing operation-specific events such as the next-element requests described above instead of generic load and store operations.

Our sparse vector store has a very simple design, and there are a number of ways to improve its microarchitecture. One option is prefetching. The sparse vector store knows the processor is doing a dot product operation and thus always knows what the processor will request next. Even a simple prefetch algorithm can be expected to reduce data delivery time.

We evaluate the Parser benchmark in a similar manner to SVM. Instead of a SPARSEVEC, we employ a type-specific storage for hash tables, HASHTAB. Similar to the SPARSEVEC, the HASHTAB at its core is simply a RAM array whose total capacity is partitioned into two regions: the first caches portions of the table backbone;

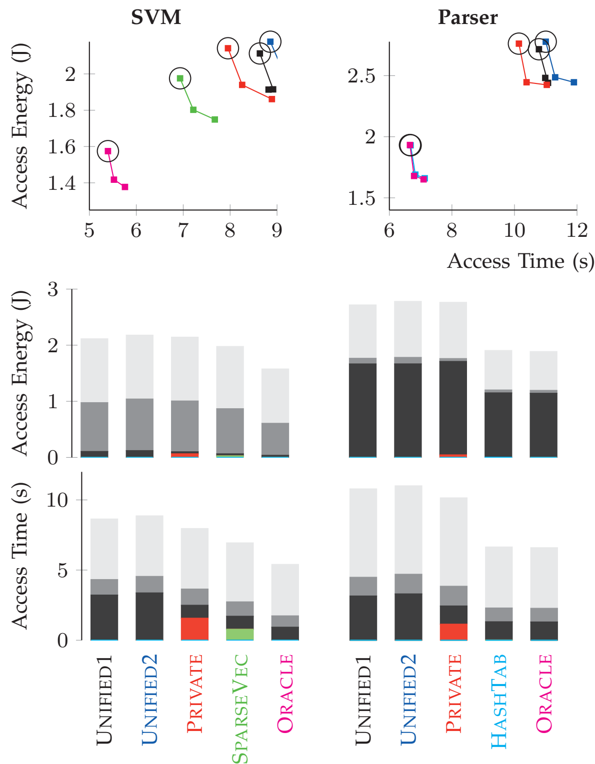

Fig. 3. Data delivery energy and performance of candidate L1 storage configurations on SVM and Parser.

the second caches table elements themselves. As with SPARSEVEC there is ample room for microarchitects to optimize the implementation of this storage structure, employing aggressive datapaths or more sophisticated storage structures such as CAMs. Other research, particularly from the networking domain, has outlined microarchitectural techniques to support efficient associative lookups in hardware [15], [4].

The scatter plots in Figure 3 (top) plot the Pareto optimal energy-performance curves for the four generic cache organizations (UNIFIED1, UNIFIED2, PRIVATE, ORACLE) plus the type-specific stores (SPARSEVEC and HASHTAB). In this case we see that the specialized store is a vast improvement over the general purpose stores, nearly matching ideal storage properties.

Specialized storage structures, SPARSEVEC and HASHTAB, showed 13–19.7% and 35.1–38% performance gains for SVM and Parser respectively while reducing energy by 5.9–8.6% and 28.9–33.1% respectively. The PRIVATE configuration is less complex to design but gained at most 7.9% and 5.9% while costing 1.4% and 1.7% more energy for SVM and Parser respectively.

Both the energy and runtime breakdowns in the rest of Figure 3 indicate that the specialized hash table store operates as a near-perfect cache. It reduces L2 pressure, which in turn reduces trips to memory, where most of the time and energy costs lie. In both cases, datatype-specific management policies were able to outperform equivalent-capacity general purpose caches, regardless of cache configuration.

7 CONCLUSION

ADPs marry high-level datatypes with processor architecture—an unusually large range of abstraction—to solve a pressing problem: how to improve the energy efficiency of large-scale computation. Our experiments found such specialization can improve instruction and data delivery energy by 27% and 38% respectively. The impact on the overall system will depend on the relative importance of instruction and data delivery, which varies between embedded systems [10] and high-performance cores [16]. ADPs represent a move away from efficient manipulation of primitive data elements to computing with higher-level types, narrowing the growing semantic gap between programmers and their hardware. This new direction for computer architecture and compilers is widely applicable to many problem domains and will empower programmers to write faster software that consumes less energy.

REFERENCES

[1] M. D. Hill and M. R. Marty, “Amdahl’s law in the multicore era,” *IEEE Computer*, vol. 41, no. 7, pp. 33–38, Jul. 2008.

[2] L. Wu, C. Weaver, and T. Austin, “Cryptomaniac: a fast flexible architecture for secure communication,” in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, Jun. 2001.

[3] H. Franke, J. Xenidis, C. Basso, B. Bass, S. Woodward, J. Brown, and C. Johnson, “Introduction to the wire-speed processor and architecture,” *IBM Journal of Research and Development*, vol. 54, no. 1, pp. 3:1–3:11, 2010.

[4] L. D. Carli, Y. Pan, A. Kumar, C. Estan, and K. Sankaralingam, “Plug: Flexible lookup modules for rapid deployment of new protocols in high-speed routers,” in *Proceedings of the Special Interest Group on Data Communication (SIGCOMM)*, Aug. 2009.

[5] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor, “Conservation cores: Reducing the energy of mature computations,” in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Pittsburgh, Pennsylvania, Mar. 2010, pp. 205–218.

[6] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, “Dark silicon and the end of multicore scaling,” in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2011, pp. 365–376.

[7] C. Jung, S. Rus, B. P. Railing, N. Clark, and S. Pande, “Brainy: effective selection of data structures,” 2011, pp. 86–97.

[8] L. Liu and S. Rus, “Perflint: A context sensitive performance advisor for C++ programs,” 2009, pp. 265–274.

[9] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, “Optimization of sparse matrix-vector multiplication on emerging multicore platforms,” 2007, pp. 1–12.

[10] W. J. Dally, J. Balfour, D. Black-Shaffer, J. Chen, R. C. Harting, V. Parikh, J. Park, and D. Sheffield, “Efficient embedded computing,” *IEEE Computer*, vol. 41, no. 7, pp. 27–32, Jul. 2008.

[11] C.-C. Chang and C.-J. Lin, *LIBSVM: a library for support vector machines*, 2001, software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm.

[12] M. R. Boutell, J. Luo, X. Shen, and C. M. Brown, “Learning multi-label scene classification,” *Pattern Recognition*, vol. 37, no. 9, pp. 1757–1771, 2004.

[13] M. Collins, “Head-driven statistical models for natural language parsing,” Ph.D. dissertation, University of Pennsylvania, 1999.

[14] “The Penn treebank project,” Online http://www.cis.upenn.edu/~treebank.

[15] F. Zane and G. Narlikar, “CoolCAMs: Power-efficient TCAMs for forwarding engines,” in *Joint Conference of the IEEE Computer and Communications Societies*, Jul. 2003, pp. 42–52.

[16] K. Natarajan, H. Hanson, S. W. Keckler, C. R. Moore, and D. Burger, “Microprocessor pipeline energy analysis,” in *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, 2003, pp. 282–287.