Stories, not Words: Abstract Datatype Processors

Martha Kim, Columbia University

I. 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, that this requires near-perfect application parallelization [7]. In practice, perfect or near-perfect parallelization is often unachievable: most algorithms have inherently serial portions and incur synchronization overheads in their parallel portions. Furthermore, parallel software requires drastic changes in 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 [10] or network processors [5] provide this for their target domain, but have usually had awkward, ad hoc interfaces that made them difficult to utilize and tended to impede software portability.

In this paper, we argue that general-purpose processors should be supplemented with abstract datatype processors (or ADPs) to deliver custom hardware performance in a form palatable to software. Abstract datatype instructions (or ADIs) implement operations on high-level data types such as hash tables, XML DOMs, relational database tables, and other types in common use in software. In object-oriented terminology, each ADI directly implements one public method of a class.

By encapsulating richer algorithms and data 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 instead of manual implementations of all functionality; ADIs simply supply such libraries in a new form.

II. Motivation

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

- *Power is paramount.* Power and cooling are projected to allow only a small fraction of transistors to be fully operational at any time [9]. With power more valuable resource than chip area, systems that incorporate specialized hardware begin to make a great deal of sense.
- *Serial performance still matters.* Recent analyses have shown that parallel speedups require near-perfect application parallelization [7]. It is imperative that we continue to investigate novel high-performance, low-power, single threaded execution techniques.
- *Do not neglect programmability.* The multicore revolution was driven by hardware needs, imposing difficult changes on software. Ten years later the technical community is still working to develop effective, reliable parallel programming techniques. On the other hand, we have a positive example of hardware/software interference in MIPS and other early RISC ISAs. The compilers that accompanied these early RISC processors shielded programmers from the attendant increases in instruction count when moving from CISC to RISC. Software technology alleviated certain challenges that might otherwise have hindered the adoption of promising hardware architectures.

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

III. Preliminary Exploration: Instruction Delivery

In this paper, we examine several potential benefits and challenges of the ADI approach. Because ADIs encapsulate algorithms that would otherwise be implemented with many fine-grained instructions, their use should greatly reduce the cost of instruction delivery. We selected 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 and cancer diagnosis. We use LIBSVM [2], a popular support vector machine library that forms the core of many classification, recognition and recommendation engines. In particular, we used LIBSVM to train an SVM for multi-label scene classification [1]. The training dataset consists of 1,211 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. We assume an ADP with support 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, freely available statistical parser developed by Michael Collins [3]. We trained the parser using annotated English text from the Penn Treebank

Project [8] 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 compare the instruction memory hierarchy in an ADI-equipped processor to its standard counterpart. We characterize the performance of the instruction memory system in two dimensions: total energy consumed (both dynamic and leakage over all levels of the hierarchy) and total time spent accessing the memory system.

To collect these data, we instrumented two applications using PIN, collected the instruction pointer stream, then fed it to our memory system simulator, which delivered statistics such as access counts for the caches and main memory, hit rates, etc. We then combined these statistics with Cacti's characterization of the access time and energy of the various cache configurations or DRAM to produce the data. For each application, we produced two instruction streams: one from an unmodified binary and that included ADIs.

Figure 1 shows the results of our instruction fetch experiments. The two columns correspond to the two applications. The top row depicts the design space exploration we performed in order to identify a set of Pareto optimal cache configurations for the subsequent experiments. To compute this data, we examined the performance of fifty-four cache configurations for ADI-enhanced and ADI-free instruction streams. We considered direct-mapped and 2-way icaches of sizes 2KB, 4KB, 8KB, ..., 512KB (each with 32B lines), and unified L2 caches (each with 64B lines) of sizes 1MB (4-way), 2MB (8-way), and 4MB (8-way). We used a fixed memory size of 1GB.

Figure 1 (top) also shows the results of our instruction fetch experiments for each benchmark: SVM (left) and Parser (right). In each plot, the first set of datapoints (—+—) shows the instruction cache behavior for ADI-free programs, while the second (—o—) shows the change in efficiency with the addition of ADIs. From this design space we select a set of Pareto optimal design points for a deeper dive to understand efficiency characteristics as we increase cache sizes.

SVM (Figure 1, left) shows the greatest improvement: a 48% reduction in access time and 27% reduction in energy consumption, re-highlighting the outsized 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.

From the fifty-four cache configurations we tested, we chose three or four Pareto optimal points for further characterization. The rest of Figure 1 depicts the a detailed breakdown of energy (Figure 1, second row), access counts (Figure 1, third row), and total instruction fetch time (Figure 1, bottom row). Each bar corresponds to a Pareto optimal cache configuration, with adjacent bars alternating between the ADI-free (baseline) and ADI-enhanced cases.

Delving into the energy consumption (Figure 1(second row)), the energy consumption is broken down into static and dynamic components for each level of the hierarchy. When

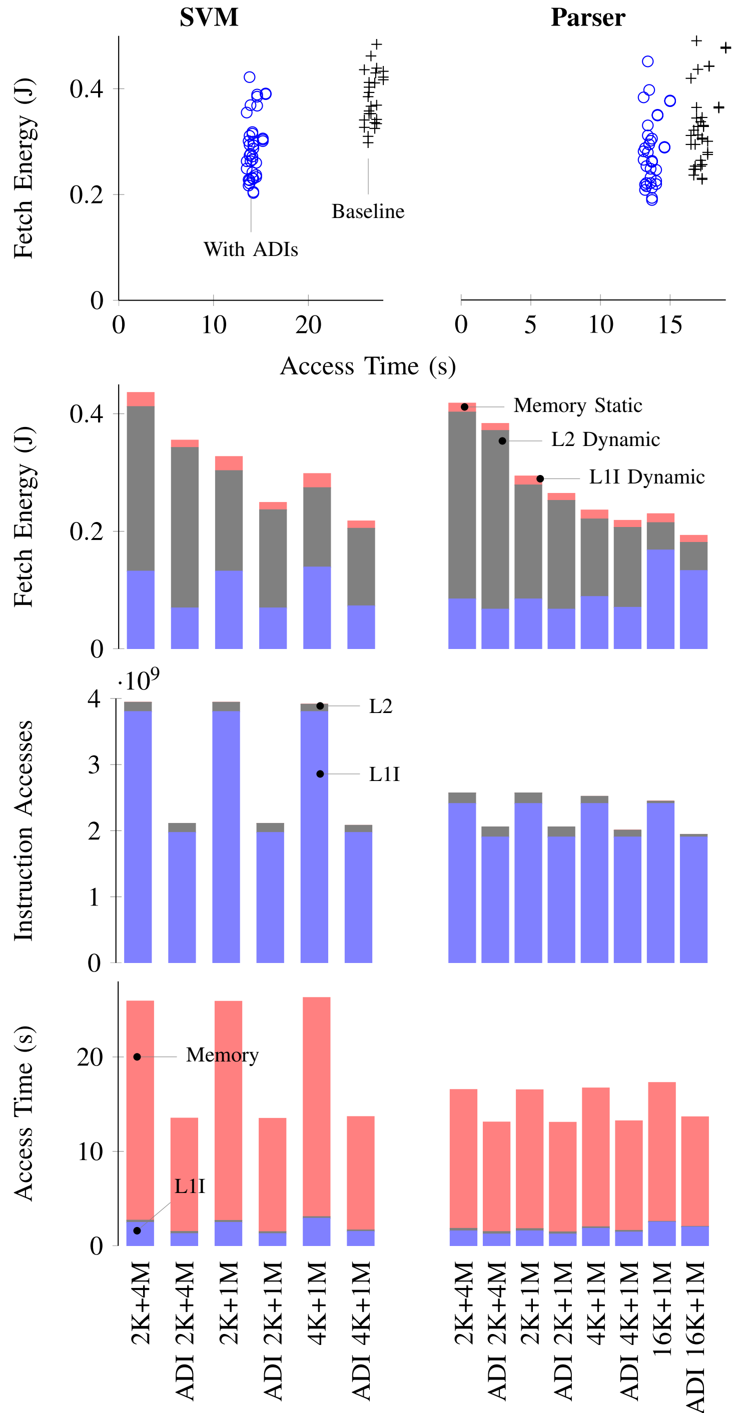

Fig. 1. Instruction fetch performance on the benchmarks

a program's instruction footprint does not fit into the L1I cache (as when the L1I cache is 2KB or 4KB), L2 dynamic energy dominates. However, the dynamic L1I energy becomes dominant as the L1I size increases and the working set begins to fit. In these plots, we have omitted bars for memory dynamic energy, L1I and L2 cache static energy, as they were roughly a thousand times smaller and can not be seen on the graph.

The above analysis is further confirmed when we examine the number of instruction accesses per level of the hierarchy (in Figure 1(third row)). As expected, L1I accesses greatly outnumber accesses at other levels. The L1 hit rate varies across benchmarks, but in all cases, the number of memory accesses are minuscule compared with the other counts.

Finally, Figure 1(bottom row) depicts the impact ADIs have on the overall time the memory system spends delivering instructions. Here, the results parallel those for the number of instruction fetches, but for a very different reason. Because main memory is so slow relative to the L1 cache, even the minuscule number of instruction fetches that miss the caches and go to main memory tend to dominate the time spent accessing memory.

To summarize our findings on instruction fetch, SVM gets the most performance gain at 48% and the most total energy reduction at 18.6%, 23.9%, and 27% for the Pareto optimal design points. Parser showed a 20.8% improvement in instruction access time and 7.5%-16% reduction in energy consumption respectively.

IV. Preliminary Exploration: Data Delivery

Recent studies have shown that instruction and data fetch dominate the time and energy consumption of general purpose computation [6], [4]. Because ADIs envelop data as well as algorithms, their implementation can employ special-purpose storage structures, that when tightly coupled with a customized datapath, could become considerably more efficient than the general-purpose alternative. While there has been a great deal of research on specialized datapaths and computation, there has been substantially less evaluation of how to extend that specialization into the memory system. In the following series of experiments we evaluate the costs and benefits of segregating and serving streams of data accesses on a type-specific basis.

We will evaluate how effectively type-specific storage structures can be deployed to service the type-specific stream of memory requests. To hold resource usage at comparable levels, we begin with a single baseline cache hierarchy (Baseline). We then augment the hierarchy with a fixed amount of additional L1 storage. This additional storage can be employed in several ways: expanding the L1D cache, adding more sets and holding the ways fixed (CacheX), expanding the L1D holding the sets constant and adding ways (CacheY), adding a second identical, private L1D cache (Private), or adding a adding an appropriate type-specific storage unit (SparseVecand HashTab). To provide an upper bound on the data access savings one can hope to see, we also model

Baseline	X, L, W	
CacheX	$2X, L, W$	
CacheY	$2X, L, 2W$	
Private	X, L, W	X, L, W
SparseVec	X, L, W	X, L
HashTab	X, L, W	X, L
Oracle	X, L, W	∞

Fig. 2. Storage architectures under comparison. Each of the five organizations is an extension of a baseline general purpose cache hierarchy, where the L1 data cache is an X byte cache, with L byte blocks, and W way associativity. Except for Oracle, each of the extensions extends Baseline, with an additional X bytes of storage at the L1 level. The Oracle configuration models an imaginary perfect memory module that can serve all accesses instantaneously and with no energy. It provides an upper bound to the improvements one can expect to see.

an infinite, instantaneous, zero-energy storage unit to serve type-related memory requests (Oracle).

A. SVM and Sparse Vector Store

We model a naïve implementation of a sparse vector store, which consists of a RAM storage array and a small number of registers which 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.

The datapath to custom storage interface can, and should, be specialized as much as possible to the target datatype. The experimental results we will show indicate that there is significant benefit both in data delivery speed and energy consumption when the computation engine can issue operation-specific events such as the next element requests described above instead of generic load and store operations.

The sparse vector storage design is painfully simple. There are a number of ways to improve the 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 naïve prefetch algorithm can be expected to reduce data delivery time.

B. Parser and Hash Table Storage

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 allocations. The

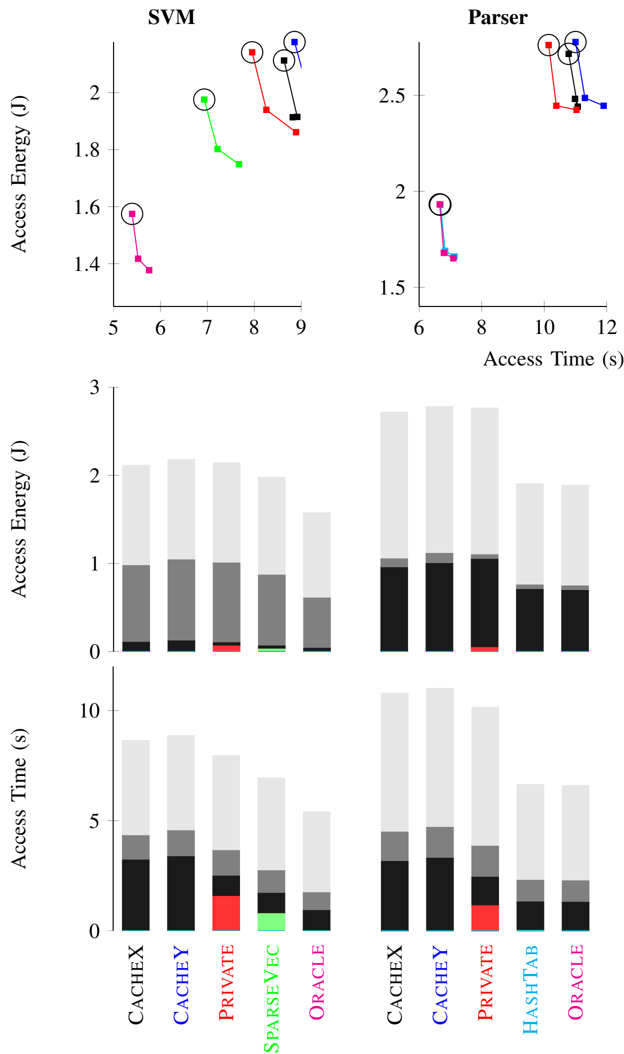

Fig. 3. Data delivery energy and performance on two target applications. The top row of plots indicates the energy-performance tradeoffs for several different L1 data storage configurations, each with equivalent resources. The bar charts display the breakdown of energy and access time by memory structure for each configuration for a single corresponding point.

first allocation is dedicated to caching portions of the table backbone, while the second allocation caches table elements themselves. As with SparseVec there is ample room for microarchitects to optimize the implementation of this storage structure.

Figure 3(right) plots the energy-performance tradeoff curves for the four generic cache organizations (CacheX, CacheY, Private, Oracle) plus the type-specific store (HashTab). In this case we see that the specialized store leaves the general purpose stores in the dust, nearly matching ideal storage properties. The energy and runtime data indicate that the specialized hash table store operates as a near-perfect cache, reducing L2 pressure, which in turn reduces trips to memory, where most of the time and energy costs lie.

V. Conclusion

ADPs marry high-level data types with processor architecture—an unusually large range of abstraction—to solve a pressing problem in computer science: improving the energy efficiency of large-scale computation. With respect to architectural design and hardware-software interfaces, ADPs represent a move away from efficient manipulation of primitive data elements to computing in terms of high-level types, narrowing the growing semantic gap between programmers and their hardware. The result will be a new direction for computer architecture and compilers widely applicable to many problem domains, and empowering programmers to write faster software that consumes less energy.

References

[1] M. R. Boutell, J. Luo, X. Shen, and C. M. Brown. Learning multi-label scene classification. *Pattern Recognition*, 37(9):1757–1771, 2004.

[2] 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`.

[3] M. Collins. *Head-Driven Statistical Models for Natural Language Parsing*. PhD thesis, University of Pennsylvania, 1999.

[4] 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*, 41(7):27 –32, jul. 2008.

[5] 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*, 54(1):3:1 – 3:11, 2010.

[6] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz. Understanding sources of inefficiency in general-purpose chips. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 37–47, New York, NY, USA, 2010. ACM.

[7] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *IEEE Computer*, 41(7):33–38, July 2008.

[8] The Penn treebank project. Online `http://www.cis.upenn.edu/˜treebank`.

[9] 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)*, pages 205–218, Pittsburgh, Pennsylvania, Mar. 2010.

[10] 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)*, June 2001.