*Appears in ACM Conference on Computing Frontiers*
*Ischia, Italy, May, 2007*

# Unified Microprocessor Core Storage

Albert Meixner
Dept. of Computer Science
Duke University
albert@cs.duke.edu

Daniel J. Sorin
Dept. of Electrical and Computer Engineering
Duke University
sorin@ee.duke.edu

## Abstract

The organization and management of microprocessor storage structures (e.g., L1 caches, TLBs, etc.) is critical to the performance and energy consumption of the microprocessor. We propose and develop the first microprocessor that can dynamically allocate storage to the structures that need it. First, we replace each existing structure with a dedicated micro-cache (µcache) that is smaller than is typical for that structure. With the smaller sizes, these structures can be made faster and less energy-hungry than the original full-size versions. Second, we back up all of the µcaches with a single Unified Core Storage (UCS). Storage in the multi-banked UCS is dynamically allocated, which alleviates performance bottlenecks. The primary benefits of UCS are a significant reduction of storage structure energy (36% less on average) and a modest improvement in performance (9.5% speedup on average).

## Categories and Subject Descriptors

C.1.1 **[Processor Architectures]** Single Data Stream Architectures - *RISC/CISC, VLIW architectures*

## General Terms

Performance, Design, Experimentation

## Keywords

microarchitecture, unified caching, power-efficiency, resource allocation

## 1. INTRODUCTION

A typical microprocessor core contains many storage structures, including caches, register file, branch history table, branch target buffer, TLBs, etc. The organization and management of these storage structures is critical to the performance and energy consumption of the processor. To the best of our knowledge, all existing microprocessors statically allocate resources to each storage structure. Some recent research has allowed specific storage structures, including caches and reorder buffers, to dynamically resize themselves for energy management purposes [3, 37, 38, 5, 13, 25, 7, 2, 10], but no scheme has allowed sharing of storage across structures. Because storage structures are often larger than they need to be to prevent them from being bottlenecks at any time, there is available storage to be dynamically reallocated. Conversely, there are situations in which a given structure would no longer be the bottleneck if it had more storage. For example, sometimes the BTB would benefit from being larger and the L1D would not suffer from being smaller. Figure 1 shows the memory utilization of various structures, assuming a unified and shared 128 KB of microprocessor storage for different SPEC2000 benchmarks. The distribution of resources among structures illustrates the different characteristics of the benchmarks. *ammp* has a large working set that can benefit from a big TLB and L1D. For *bzip*, all structures except the L1D can be made small, because most time is spent is a tiny section of code and memory is accessed sequentially. *gcc-scilab* requires a large L1I, BHT, and BTB, because of its big code footprint. *gcc-166* does not share *gcc-scilab*'s requirements, which suggests that resource utilization depends not only on the application but also on the workload.

In this work, we propose and develop the first microprocessor that can dynamically allocate storage among its structures. The storage structures we incorporate into our scheme are: L1I and L1D caches, I-TLB and D-TLB, branch history table (BHT), and the branch target buffer (BTB). First, we replace each of these structures with a dedicated *micro-cache* (µcache) that is smaller than is typical for that structure. Second, we back up all of the µcaches with a single, multi-banked *Unified Core Storage (UCS)*. The smaller, complexity-effective [24] µcaches can be made less

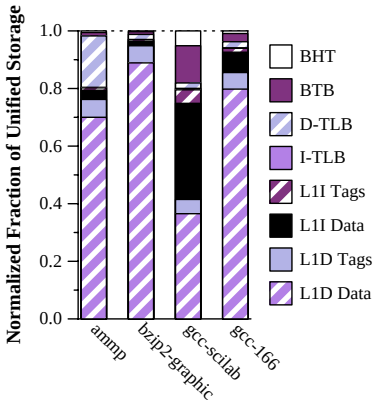


**Figure 1. Time-averaged utilization of 128K shared storag**



*CF'07*, May 7-9, 2007, Ischia, Italy.

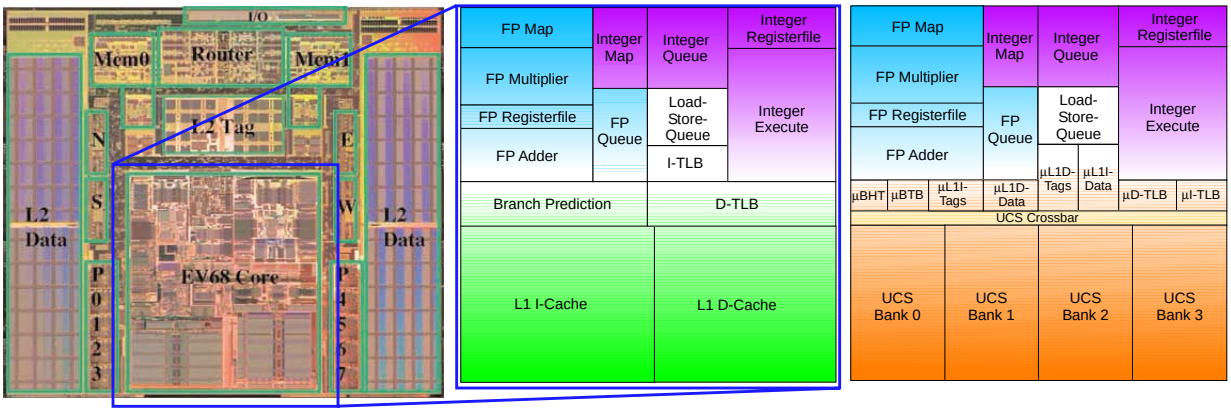


**Figure 3a. Alpha 21364 Die Photo**
*From Krewell [18]*

**Figure 3b. Alpha Core Floorplan**
*From Skadron et al. [33]*

**Figure 3c. UCS Core Floorplan**
*UCS component sizes estimated*

**Figure 3. Floorplans for conventional and UCS processors**

energy-hungry and sometimes faster than the original full-size structures. The µcaches also allow all structures to share the UCS without undue contention.

Our approach is similar to existing multi-level structures (e.g., for register files or branch predictors) that use a small cache backed up by dedicated lower level storage [31, 35, 39, 8, 29, 14, 6, 1, 19], except the UCS *unifies the lower level storage across all structures*. This unification allows the UCS processor to improve storage utilization compared to having a separate lower level storage for each structure. Furthermore, by dynamically allocating storage in the UCS, the UCS improves the overall hit rate (in the µcache or UCS) for all of our structures and dynamically alleviates performance bottlenecks.

The primary benefits of the UCS processor, with respect to a baseline processor without UCS, are:

- **Reduced energy consumption**: The UCS processor uses 64% of the storage structure energy (including the L2 cache) of the baseline processor. Since storage structures use a large fraction of a processor's energy [22], the energy reduction achieved by UCS constitutes a significant savings.
- **Improved performance**: The UCS processor has an average speedup of 9.5% (across the SPEC2000 benchmarks) over a comparable processor with statically allocated structures.
- **Storage efficiency**: UCS allows more efficient use of the physical storage space in the structure and can achieve performance equal to a conventional processor with less storage space (see Section 4.2.4).

The remainder of the paper first presents an overview of how to design a microprocessor with UCS in Section 2. In Section 3, we discuss the designs of the µcaches for each storage structure. Section 4 contains an experimental evaluation of our UCS processor and compares it to a typical microprocessor with statically allocated storage. In Section 5, we discuss related work, and we conclude in Section 6.

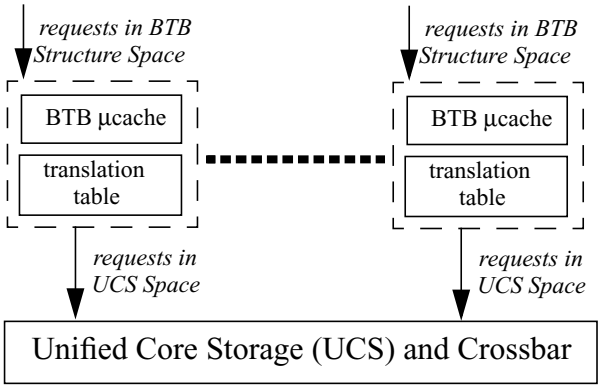


**Figure 2. High level view of the UCS architecture**

## 2. UCS MICROPROCESSORS

At a high level, the UCS portion of the processor looks like Figure 2. Since many µcaches and the L2 cache all contend for access to the UCS, we connect them to the UCS with a crossbar. Similar crossbars have been used extensively for multi-bank caches [30] and have also been proposed as a means of cache sharing in CMPs [21]. Dutta et al. [9] have analyzed crossbar designs that closely match UCS requirements, and they have found the required area surprisingly small for crossbars with fewer than 32 ports. In 250nm technology, a 32-bit wide 8-port crossbar requires 1.60mm$^2$ and can be clocked at up to 1GHz. In UCS processor, we assume a wider crossbar (8-port, 64-bit), which will increase area requirements. This increase is likely to be offset by the smaller assumed feature size of 90nm. Wang et al. [36] found the energy for one flit traversal of an 8-port 128-bit router (crossbar + input buffers) in 100nm technology to be about 0.06 nJ or about 25% of the energy to read 128 bits from a 16K SRAM, according to *Cacti* [15].

Independent of its area and latency, the layout of the crossbar is a major concern, because it connects several units used in different pipeline stages to a large shared storage structure. By positioning units appropriately, we can layout the crossbar without adding long

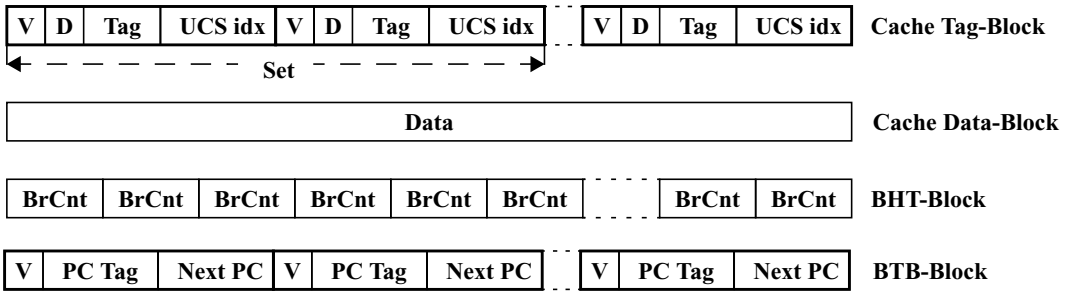


**Figure 4. Contents of blocks for various structures**

**Table 1. Storage structures attached to the UCS and equivalents in the baseline processor**

| Structure | Baseline Processor | UCS Processor | |
|---|---|---|---|
| | | virtual | µcache(s) |
| **L1I cache** | 64KB, 2-way set-assoc<br>3 cycles | 128KB, 2-way set-assoc | tags[a]: 16 blocks, 2-way set-assoc<br>data[a]: 16 blocks, direct-mapped |
| **L1D cache** | 64KB, 2-way set-assoc<br>3 cycles, 2 R/W ports | 128KB, 2-way set-assoc | tags[a]: 16 blocks, fully-assoc, 2 R/W ports<br>data[a]: 16 blocks, fully-assoc, 2 R/W ports |
| **I-TLB** | 16 entries, fully-assoc | 4K entries, 2-way | 8 blocks, 2-way |
| **D-TLB** | 32 entries, fully-assoc | 4K entries, 4-way | 16 blocks, 2-way |
| **L2-TLB** | 512 entries, 4-way | *N/A* | *N/A* |
| **BHT** | GShare with 16K entries, 14-bit history | GShare w/128K entries, 11-bit history | 8 blocks, direct-mapped |
| **BTB** | 2K entries, 4-way, 2 cycles | 4K entries, direct-mapped | 8 blocks, direct-mapped |
| **RUU** | 128 entries | 1K entries[b] or *N/A* | 128 entries |
| **Shared Storage** | *N/A* | 128KB direct-indexed, 8 banks, 1 port/bank, 3 cycles (incl. arbitration) | |
| **Total bytes** | ~150KB (including tags) | *N/A* | ~139KB (128K UCS+7K µCaches+4K BAT) |

a. We will discuss the access latencies for the L1I and L1D µcaches in Section 3.1.

b. As explained in Section 3.4, we do not always include the RUU in our scheme.

wires across the chip. Figure 3a shows a die photo of the Alpha 21364, which we use as an example. Figure 3b (from [33]) shows where different units are located within the Alpha core. In Figure 3c, all major storage structures have been replaced with UCS µcaches and we have added the shared UCS storage and crossbar. The figure is not intended to show exact area and layout of the UCS components, but to give an intuition of the their positioning within the core. Although no changes were made to the layout of the remaining core, all the µcaches are adjacent to the crossbar and close to the locations of the conventional storage structures they replaced. This ensures that the crossbar can be laid out efficiently and the µcaches can be easily connected to the unmodified units.

All µcache and UCS blocks are 64 bytes, and a block can hold different entries depending on the structure using the block as illustrated in Figure 4. We only consider inclusive UCS designs (i.e., every block in a µcache is also in the UCS) to simplify writebacks from µcaches to the UCS. To translate from each structure's individual address space (*Structure Space*) to the UCS's shared address space (*UCS Space*), we add a per-structure *Translation Table (TT)*. The UCS is directly indexed using the address found in the TT.

## 2.1 Baseline System Model

Our baseline system is a typical dynamically scheduled processor. It has a simple, unified RUU [34], but this design decision does not affect the discussion or results. We assume the L1 caches are virtually indexed and physically tagged. We do not include the L2 cache (or beyond) in the UCS scheme. We also do not include the instruction fetch queue, reservation stations, or load-store queue, for reasons we discuss in Section 3.4. Table 1 provides the configuration of the storage structures included in the UCS scheme for both the baseline and UCS processors (discussed next), and Table 2 provides the configuration parameters common to the baseline and UCS processors. The processor configurations are modeled loosely after the AMD Athlon XP. We chose the UCS configuration such that it has roughly the same total amount of storage as the baseline system, although results show that the UCS

**Table 2. Common Processor Parameters**

| Feature | Configuration |
|---|---|
| Pipeline depth | 10 stages |
| Pipeline width | fetch, decode, commit: 4; issue: 6 |
| Functional units | 4 Integer ALUs, 1 Integer Mult/Div, 4 FP ALUs, 1 FP Mult/Div |
| L2 cache | 1MB, 4-way set-associative, 8 banks, 11 cycles |

processor's performance is relatively insensitive to having less storage.

## 2.2 Structure Virtualization

Each individual structure in a UCS processor is designed as if it was implemented using private, fixed-size, flat storage. To enable storage sharing, we reinterpret the addresses of data items in the private structures as virtual addresses in *Structure Space*. Since *Structure Space* does not correspond to actual physical memory, these addresses have to be translated to *UCS Space* using a scheme that is similar to virtual memory (VM)[1]. An address translation uses the per-structure TT, which is a lookup table directly indexed using the upper bits of the *Structure Space* address. The TT entry contains the location of the requested block in *UCS Space*. TTs are cheap and fast because they are small, un-tagged, single-ported, and non-associative. All operations perform in the µcaches and thus TT translations are only necessary during µcache refills and writebacks from µcaches to the UCS. The address range of the *Structure Space* is determined by the *virtual* size of the structure, e.g., a 16K-entry BHT with 2-bit history counters has a 4KB *Structure Space*.

Virtualization allows a UCS processor to shift storage from under-utilized structures to highly stressed ones and therefore we can increase the (virtual) size of many structures without requiring additional physical storage [23]. The larger virtual structures often suffer fewer capacity misses. For especially time-critical structures such as the BTB, we can replace a highly associative structure with a much larger, in terms of virtual size, direct-mapped one and avoid the latency and energy costs of high associativity without inflicting additional conflict misses at the cost of an extra level of indirection.

## 2.3 UCS Design

There are several design issues for the UCS, including size, banking, and the policy for replacing UCS blocks. We assume for now the UCS configuration in Table 1: 128KB, direct-mapped, with 8 single-ported banks and a UCS block size of 64 bytes. Based on experiments with Cacti [15], we assume that the UCS has a 3-cycle access latency, including arbitration for the crossbar; we experimentally show in Section 4 that the UCS processor's performance is insensitive to UCS access latencies even as long as 5 cycles.

1. A structure's private *Structure Space* is the equivalent of a process' private virtual address space in standard VM, and *UCS Space* is analogous to the physical address space. The translation table (TT) is analogous to the page table.

**Table 3. Utility Values.**
*Hit and miss rates obtained via profiling.*

| | Utility value | Function used to determine value |
|---|---|---|
| L1I data | 8 | (L2 hit rate)*(L2 hit latency) + (L2 miss rate)*(memory latency) |
| L1I tags | 128 | #L1I tags per UCS block) * (L1I data utility) |
| ITLB | 240 | (#ITLB entries per UCS block) * (ITLB refill latency) |
| BHT | 3072 | (#branch counters per UCS block) * 2 * (branch misprediction penalty) |
| BTB | 96 | (#BTB entries per UCS block) * 2 * (branch misprediction penalty) |
| L1D data | 2 | [(L2 hit rate)*(L2 hit latency)+ (L2 miss rate)*(mem latency)]/(pipe width) |
| L1D tags | 64 | 2*(#L1D tags per UCS block) * (L1D data utility) |
| DTLB | 240 | (#DTLB entries per UCS block) * (DTLB refill latency) |

The replacement policy is the most challenging issue and must consider three aspects: ranking blocks based on utility, finding blocks with low utility, and then selecting from among those low-utility blocks. All information necessary for UCS block management resides in the *block allocation table (BAT)*, which contains one entry for each block in the UCS. For a block currently allocated to one of the CPU structures, the BAT entry holds the identity of that structure and the 12-bit *utility value* of the block. The BAT entry for a block that is not currently allocated to any structure holds the index of the next free block, thus forming a singly-linked *free list* of unallocated blocks.

**Ranking Blocks by Utility.** In a standard cache, the utility of a block can be reliably estimated from its access frequency. In UCS, the content type of the block (e.g., L1I data, or BHT entries) must also be considered in replacement decisions. To compare blocks of different types we define the utility value as the estimated number of cycles the processor will stall if the block is evicted. Determining exact utility values would require detailed statistics of past system behavior. Instead we rely on statically estimated utility values, listed in Table 3, which provide good results at a much smaller cost. Moreover, our results are not very sensitive to these exact values. A block is assigned its base utility value whenever it is fetched into a µcache. While the block is idle, its utility value decays (described next), similar to Cache Decay [16].

**Finding Low-Utility Blocks Quickly.** To quickly find a block with low, though possibly not lowest, utility value, the UCS maintains a small (4-entry) *replacement candidate table*. The replacement candidate table is updated by the *UCS walker*, which scans the BAT in the background (at a rate of four entries per cycle) for blocks having utility values lower than one of the current replace-

ment candidates. If the UCS walker finds such a block, it puts the block's address into the replacement candidate table. The UCS walker also decays utility values by dividing the utility value of each inspected block by a constant that is chosen to allow division using bit shifts.

**Selecting Low-Utility Blocks for Replacement.** Whenever the free list is empty, the UCS walker selects one of the blocks in the candidate table for eviction. The block owner can refuse to free the block suggested by the UCS walker, but it must free a block allocated to it. Allowing the structure to decide on its own block evictions is important in certain situations. For example, the L1I and L1D have separate tag and data blocks, and each tag block contains a pointer to a data block that is useless if its tag block is replaced.

# 3. DESIGNING μCACHES FOR SPECIFIC STRUCTURES

We now discuss the designs of specific structures, since the μcache designs vary based on the type of structure. We base all of our timing and energy assumptions for the shared storage, μcache, and the crossbar on experiments using Cacti [15]. Crossbar energy consumption is based on the energy used by the routing network of an interleaved cache. We found the resulting values to be consistent with work that specifically addresses on-chip crossbars and routers [9,36]. In Sections 3.1 through 3.3 we describe our designs for the L1 caches, branch predictor structures, and TLBs. In Section 3.4, we discuss why we have not included certain structures in the UCS scheme.

## 3.1 L1 Caches

We design our L1 caches to take advantage of the locality in small working sets, while not sacrificing much performance for accesses outside of these working sets. Similar to a filter cache [17], the L1 μcaches reduce energy consumption of accesses to the most popular addresses. We improve cache utilization compared to the baseline system by dynamically allocating space in the UCS to the L1I and L1D μcaches. This differs from simply having a unified L1 in three ways: instructions and data have separate tag arrays to reduce conflict misses, no additional L1 read ports are required to access code and data, and high-penalty L1I misses are avoided at the cost of a lower L1D hit rate due to different utility values.

The basic design of our L1 μcaches is inspired by the V-Way Cache [27], which logically separates the tag array from the data array. Both of our L1 μcaches (L1I and L1D) are split into an L1-μtags and L1-μdata, but our L1 μcaches differ from the V-Way Cache in four important ways. First, to reduce access latency and energy consumption, the L1-μtags and L1-μdata are just small (16-entry) filter caches. Second, by virtualizing the tag array, we allow tags to be physically located anywhere in the UCS and we eliminate the need to store inactive sets. Third, we replicate the tags in the L1-μdata, in order to allow low-latency L1 μcache accesses during periods with high locality. Fourth, an entry in the L1-μtags contain the index of a data block in the UCS as well as an index into the L1-μdata if the block is currently there. Because the L1-μtags lookup provides a direct index into the UCS, no tag comparisons are needed at the UCS and UCS access can be fast. The index

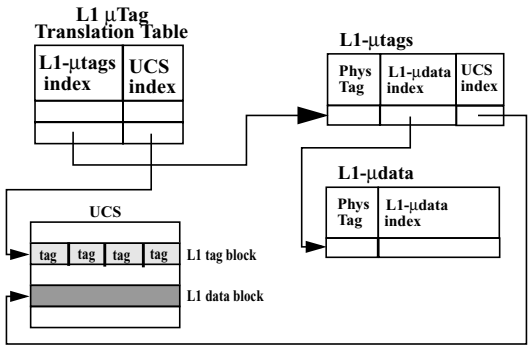


**Figure 5. Organization of L1 μcaches in UCS**
*(applies to both L1I and L1D)*

obtained during L1-μtags lookup specifies an index in UCS Space, not an address in Structure Space, and does not have to be translated. Therefore L1-μdata does not need a TT (but the L1-μtags has a TT).

The organization of our L1 μcaches, as shown in Figure 5, gives us several options for how to access them. We leverage the different energy and performance characteristics of these access modes to dynamically switch between them depending on the situation. We now describe how the L1I and L1D μcaches use these different access modes, and then we discuss UCS's impact on cache coherence.

### *3.1.1 L1I μCache*

L1I cab operate in one of two modes (*fast* and *slow fetch mode*) depending on the state of the pipeline.

**Fast Fetch Mode.** The latency of a fully pipelined cache serving an in-order component, such as the fetch stage, is only important when the pipeline is restarted, most commonly after a mispredicted branch. To allow fast recovery from mispredictions, the L1I operates in *fast fetch mode* after each pipeline flush. In *fast fetch mode* (Figure 6a), the L1I-μdata operates as a tagged, direct-mapped cache that is accessed in parallel with both the L1I-μtags and the I-TLB μcache. In the first cycle, only a partial tag match in the L1I-μdata is made using the page offset bits of the virtual address. The returned data is used speculatively while a full tag match is performed in the second cycle. *Fast fetch mode* results in low cache latency but unnecessary energy consumption during data L1I-μdata misses. Since the benefit of the low initial latency disappears with the first L1I-μdata miss, such a miss triggers a switch to *slow fetch mode*.

**Slow Fetch Mode.** In *slow fetch mode* (Figure 6b), the L1I-μtags and L1I-μdata are accessed sequentially to conserve energy. The L1I-μtags is still accessed in parallel with the I-TLB μcache, but it requires a full physical address before the correct way can be determined. At this point, the location of the block in the UCS is known, as well as if and where it is in the L1I-μdata. Although having this information would allow us to place blocks anywhere in the L1I-μdata, we still use a direct-mapped L1I-μdata to be able to switch back to *fast fetch mode*. After the L1I-μtags access, the block is read from either the UCS or the L1I-μdata, which are both direct SRAM accesses without tag comparisons.

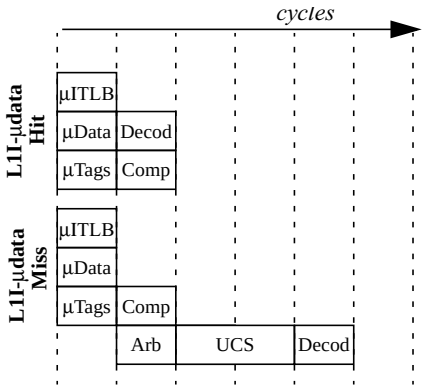


**Figure 6a. UCS L1I Fast Fetch Mode. Assumes L1I-µtags hit.**

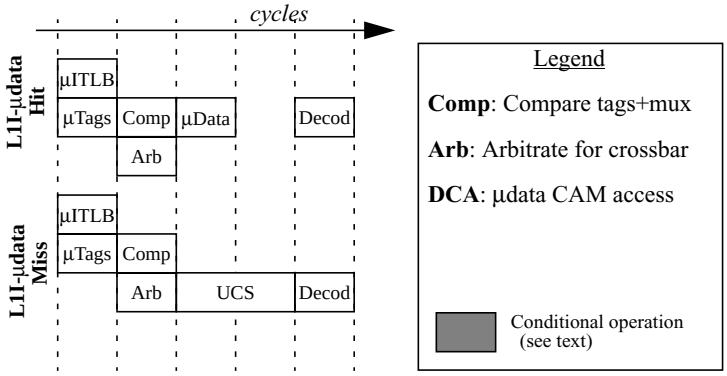


**Figure 6b. UCS L1I Slow Fetch Mode. Assumes L1I-µtags hit.**

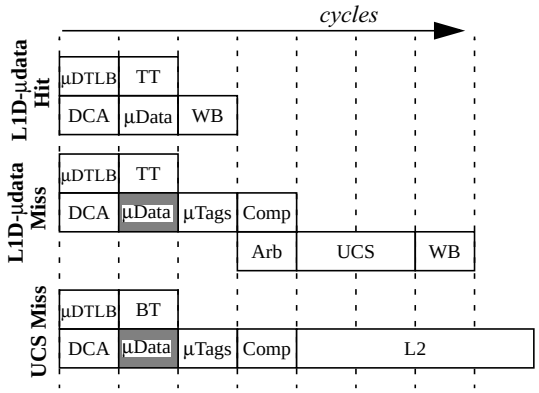


**Figure 6c. UCS L1D Access**

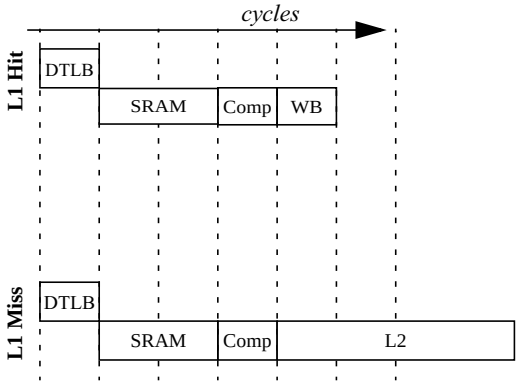


**Figure 6d. Standard (non-UCS) L1D Access**

### *3.1.2 L1D µCache*

Instead of using a direct-mapped L1D-µdata, like we did for the L1I-µdata, we choose a fully associative design for two reasons. First, because data accesses can be performed out-of-order, the L1D hit latency has slightly less impact than L1I latency. Second, direct-mapped caches typically exhibit more conflict misses for data than for instructions. As illustrated in Figure 6c, in parallel with the D-TLB µcache lookup, the page offset bits are used to locate a block in the L1D-µdata. We use a CAM with a partial tag restricted to page offset bits for this purpose. To avoid multiple matches in the L1D-µdata CAM lookup, block placement in the L1D-µdata is restricted such that no two blocks in the cache have identical page offset bits. This restriction can easily be enforced during block replacement since colliding blocks will be found during the cache miss. Because the CAM only uses a partial tag, a full tag comparison in the following cycle is necessary to determine if the request hit in the L1D-µdata. If no partial match is found in the L1D-µdata or the full tag match in the L1D-µdata fails, we search for the block in the UCS. To obtain the block's UCS index, the tag corresponding to the requested address is queried in the L1D-µtags. To find the tag block without an associative lookup, the L1D TT also holds the indices of blocks currently in the L1D-µtags[2], such that TT access provides both tag block location in the cache (if L1D-µtags hit) and the UCS (if L1D-µtags miss). After the tag block is located, a tag comparison is performed to find the correct tag within the tag block. For comparison, we illustrate an L1D access for our baseline processor in Figure 6d.

## 3.2 Branch Predictor Structures: BHT & BTB

The UCS scheme addresses the problems previously identified [14] for multi-cycle branch predictors, namely the increased delay for correctly predicted branches as well as increased branch miss latency.

**BHT.** The BHT µcache is small and direct-mapped, and it can be accessed in one cycle. It is similar to previous BHT caching [14], but it adds a modification to improve locality. A GShare predictor indexes the BHT using an exclusive-or (XOR) of the PC and branch history register, which is a shift register and does not exhibit any spatial locality. To achieve acceptable hit rates we only perform the XOR for the bits that describe the offset of a branch counter within a UCS block, such that the UCS block address is solely determined by the PC. This modification leads to a minor decline in branch prediction accuracy but a huge increase in the BHT µcache hit rate. Since both L1 and BHT are indexed using the PC almost all misses in the BHT µcache are preceded by misses in the L1I µcache. Therefore BHT blocks can be prefetched during the L1I refill. In the rare case of BHT µcache misses the fetch stage is stalled.

2. In a multi-processor, the coherence controller also needs access to the L1D TT and L1D-µtags to locate invalidated blocks.

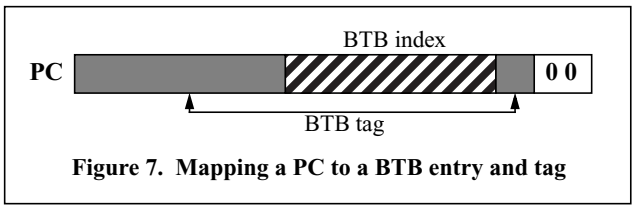


**Figure 7. Mapping a PC to a BTB entry and tag**

**BTB.** To make the BTB accessible within a single cycle, both the virtual BTB and its µcache are direct-mapped. The increase in conflict misses due to direct-mapping is mitigated by having a much larger (virtual) BTB, which also reduces the number of capacity misses in benchmarks that execute branch-heavy code with a large memory footprint. As with the BHT, we stall the fetch stage on BTB µcache misses and, although the BTB µcache can prefetch during L1I refills, a high BTB µcache hit rate is critical to UCS performance. To improve µcache locality, the BTB µcache is not indexed using the least significant bits of the PC, but rather using *n* of the least significant bits as part of the tag. That is, $2^n$ neighboring addresses map to the same index, as shown in Figure 7. We exploit the observation that back-to-back branches are infrequent and most branches are preceded by several instructions that compute the branch condition. For small *n* (e.g., *n=2* as in our experiments), this optimization only slightly increases conflict misses, but vastly improves BTB µcache locality.

## 3.3 Translation Lookaside Buffers

Standard TLBs are small structures and there is little gain in trying to shrink them even further. Therefore the TLB µcaches (for both the I-TLB and D-TLB) are roughly the same size as the original TLBs. However, because the TLB µcaches contain fragments of a larger virtual structure, they operate differently.

To find a page translation in an *m*-way set-associative virtual TLB that is cached in an *n*-way set-associative TLB µcache, *m* locations in each of *n* blocks in the TLB µcache must be checked. We organize the TLB µcaches such that all tags for these *m*n* entries are searchable using a small CAM. The TLB µcache contains multiple (typically 4-16), such CAMs, one of which is selected for lookup based on the address that is translated. Compared to a standard TLB, which has to query every entry in its CAM, our scheme can use CAMs with fewer entries. Furthermore, the CAM entries in our scheme have smaller tags (e.g., 10 bits), because entries within a set of the virtual TLB have a number of identical bits which only need to be stored once in the CAM, and other bits used to index the set are not stored in the CAM at all. Thus, our CAM organization reduces energy usage at the cost of a slightly lower TLB hit rate due to conflict misses.

We improve TLB µcache hit rates by using higher order bits for set selection (i.e., mapping consecutive pages to the same set), as we did for the BTB µcache. Unlike with the BTB µcache, this decision will not necessarily lead to conflicts if translations for two neighboring pages are present in the TLB µcache, because the TLB is not direct-mapped. Nevertheless, the gain in TLB µcache locality also comes at the cost of a slight increase in the number of misses in the virtual TLB.

## 3.4 Structures Not Included in UCS Scheme

The three non-trivial storage structures that we do not include in our UCS scheme are the instruction fetch queue (IFQ), reservation stations, load-store queue (LSQ), and register file (or RUU). Unlike the structures included in the UCS processor so far, neither the reservation stations nor the LSQ typically exhibit much locality. The LSQ requires an associative lookup over all entries during loads, while the reservation stations need access to all entries during instruction selection. Such accesses are infeasible in a virtualized structure, because only the entries residing in the µcache are directly accessible. Our experiments showed that incorporating the IFQ into the UCS scheme provides only negligible performance benefit and increases energy consumption. The register file is the best candidate for inclusion in the UCS scheme in a way similar to Oehmke et al.'s register cache [23]. An RUU is less suitable for caching than an explicitly renamed register file, because the FIFO access pattern limits locality in the accesses and the RUU is accessed by multiple units.

# 4. EXPERIMENTAL EVALUATION

The goal of our experiments is to evaluate the energy and performance characteristics of a microprocessor with UCS and quantitatively compare it to a baseline (non-UCS) microprocessor.

## 4.1 Methodology

We performed all of our experiments using a modified version of SimpleScalar [4]. All UCS structures were modelled cycle accurately, including contention for the UCS banks. For benchmarks, we used the SPEC 2000 benchmarks[3], and we sampled them using the SimPoint methodology [32]. We used Cacti [15] to compute the latencies and energy profiles of all storage structures for a 90nm process. Our experiments model the energy consumption of the crossbar to the UCS (but not the small amount of arbitration logic), BAT maintenance by the UCS walker, and accesses to the TTs.

## 4.2 Results

We first explore how the different structures share the UCS, to provide insight into the UCS processor's behavior. We then compare the UCS processor to the baseline, in terms of energy and performance. Lastly, we discuss our sensitivity analyses.

### *4.2.1 UCS Sharing Behavior*

The initial motivation for sharing storage space among different structures was the hypothesis that the storage requirement of each structure is highly workload dependent and therefore no static assignment of storage to structures will achieve full utilization of the available resources. In Figure 8, we illustrate the time-averaged UCS utilization of each benchmark. Although the L1D cache uses the most storage space, the exact distribution of resources among the various structures differs significantly between benchmarks. Since the data footprint of virtually all programs is larger than the code footprint, the L1D cache ends up grabbing all resources that

3. We could not include art, fma3d, and sixtrack due to problems with our simulation environment.

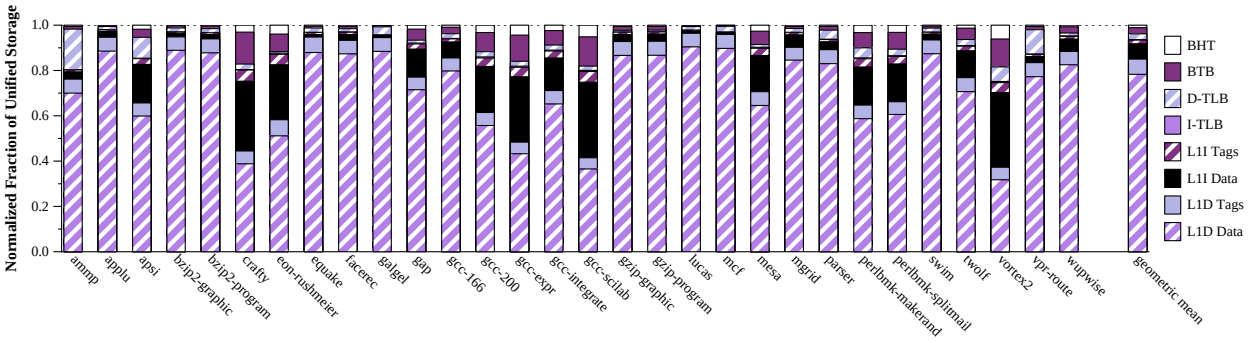


**Figure 8. Amount of storage space used by different structures normalized to the total UCS size (128K)**

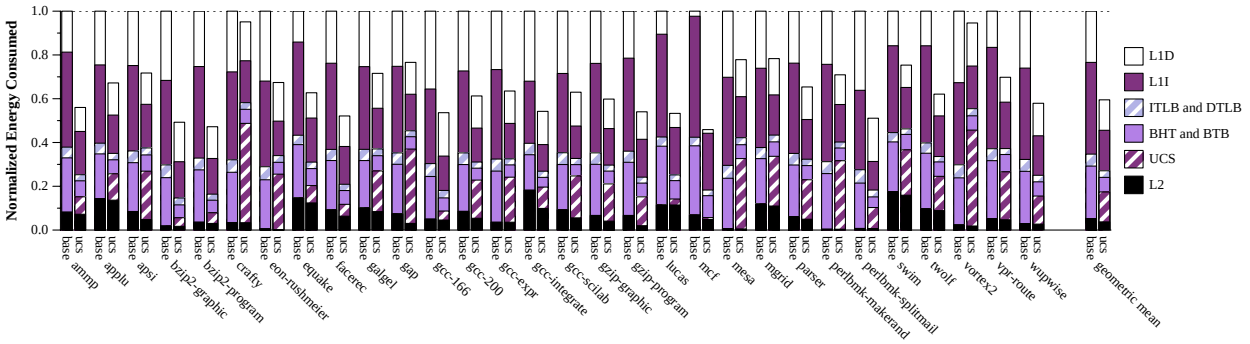


**Figure 9. Energy consumption breakdown for baseline processor and UCS processor (normalized to baseline)**

are not used by other structures even if it does not improve L1D hit rates. In this case, parts of the UCS are effectively unused and will have near-zero utility values. If we implemented a mechanism for UCS compaction or clustered allocation of potentially low-value blocks, individual UCS banks could be disabled without a performance penalty.

### *4.2.2 Energy Used by Storage Structures*

Although the UCS processor uses some extra energy for µcache misses, our goal is for that increase to be dwarfed by the energy reduction achieved for µcache hits. In Figure 9, we plot the energy consumptions of selected storage structures in the baseline processor and the UCS processor, normalized to that of the baseline processor. The results show a dramatic reduction in storage energy for the UCS processor, with an average energy consumption that is 64% of the baseline. In particular, L1I energy is far lower for the UCS processor (where L1I energy includes energy used by the L1I-µtags, L1I-µdata, and L1I-µtags' TT), and L1D energy is also significantly reduced. These energy reductions are partially offset, but never outweighed, by the energy consumption of the UCS. On several benchmarks, energy reduction over the base configuration is near or above 50%, and all benchmarks show at least some reduction in energy.

Part of the UCS processor's energy benefit derives from its better utilization of the L1 caches and thus fewer L2 cache accesses. In Figure 10, we plot the number of L2 accesses made by the UCS processor, normalized to the number of L2 accesses made by the baseline processor. On average, the UCS processor makes less than 71% of the L2 accesses of the baseline processor.

### *4.2.3 Performance*

We expect the UCS processor to outperform the baseline processor on those benchmarks that can benefit from reallocation of on-chip storage structures. The UCS processor may potentially perform slightly worse than the baseline processor for benchmarks that both do not benefit from reallocation and suffer from frequent µcache misses. Table 4 shows the average miss and fill rates for all µcaches. The majority of misses is caused by the load-store unit in the out-of-order part of the core. The more performance-critical µcaches in the in-order front-end miss infrequently.

In Figure 10, we plot the runtime of the UCS processor, normalized to the baseline processor. The mean speedup is a modest 9.5%, with some speedups as large as 30%. There are 5 slowdowns, the worst of which is 6%, and no other slowdown exceeds

**Table 4. µcache miss and fill rates**

| Structure | Misses[a] | Fills | per |
|---|---|---|---|
| **BHT** | 0.002 | 0.044 | branch |
| **BTB** | 0.012 | 0.143 | branch |
| **L1I tags** | 0.003 | 0.004 | instruction |
| **L1I data** | 0.032 | 0.037 | instruction |
| **L1D tags** | 0.052 | 0.052 | memop |
| **L1D data** | 0.157 | 0.157 | memop |
| **ITLB** | 0.001 | 0.001 | instruction |
| **DTLB** | 0.016 | 0.016 | memop |

a. Prefetches count as *Fills*, but not *Misses*

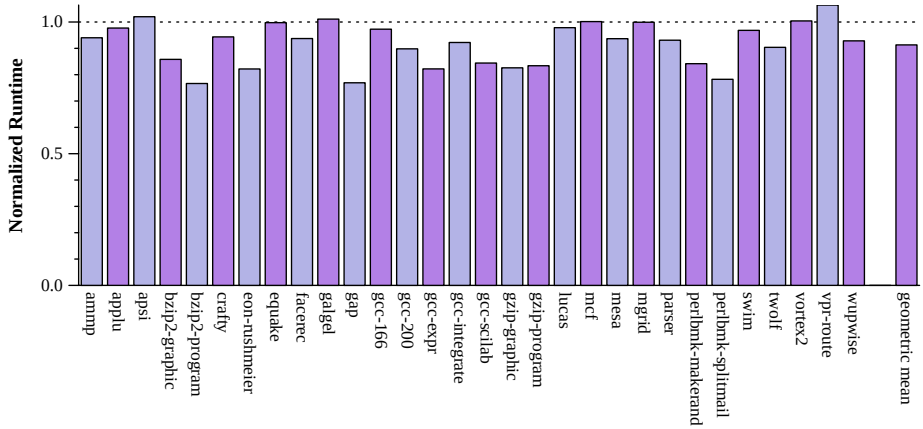


**Figure 10. UCS processor runtime (normalized to baseline processor)**

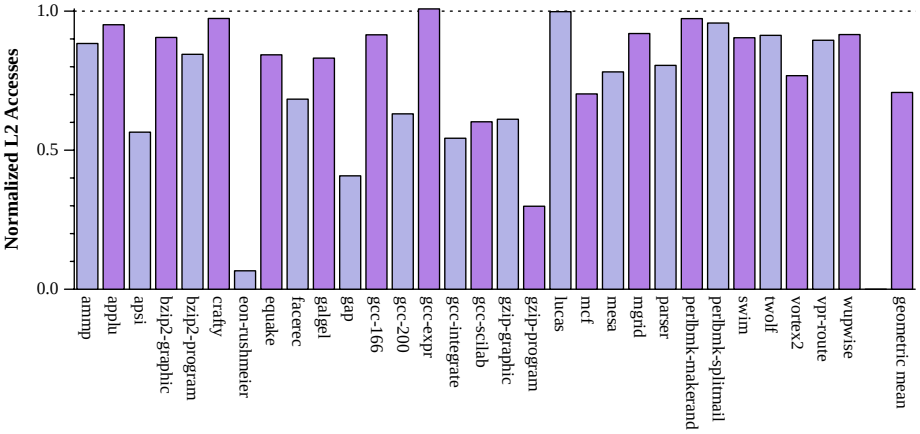


**Figure 11. L2 cache accesses by UCS processor (normalized to baseline processor)**

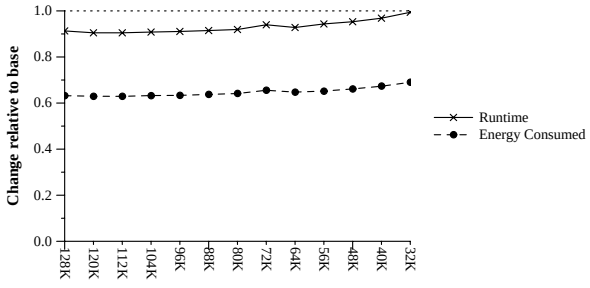


**Figure 12a. Sensitivity to UCS size**

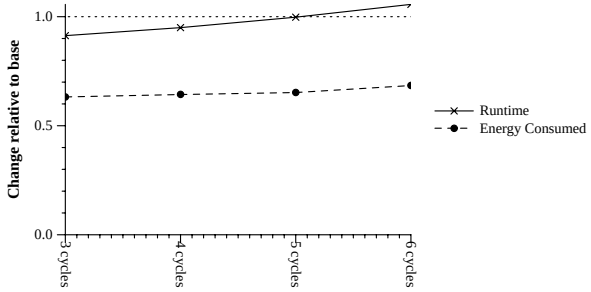


**Figure 12b. Sensitivity to UCS access latency**

2%. The reasons for the observed speedups differ from benchmark to benchmark, depending on the structures most stressed by the workload. The compression benchmarks *gzip* and *bzip* spend most of their execution time in a small loop and can therefore dedicate almost the entire UCS storage space to the L1D to avoid L2 accesses (see Figure 11). The branch intensive benchmarks gain performance due to the low latency branch prediction µcaches and *fast fetch mode* after mispredicted branches. Some benchmarks, like *perl*, also benefit from the larger, virtual BTB, which reduces the number of incorrectly predicted target addresses. In *gap*, the benchmark with the highest L1I miss rate in the baseline processor, the virtual L1I cache eliminates almost all instruction cache misses and thus reduces fetch stalls.

All benchmarks that experience a slowdown on the UCS-based processor incur frequent misses in the L1D-µtags, which increases the average load latency. This increased load latency then leads to higher occupancy in the LSQ and reservation stations, which consequently causes resource stalls.

### *4.2.4 Sensitivity Analysis*

We explore the UCS design to ensure that it is robust and not overly sensitive to small changes either in its design or latency. In Figures 12a and 12b, we independently vary the UCS's size and latency, and we plot the UCS processor's runtime and energy consumption. Individual datapoints are averages across all benchmarks, and datapoints are normalized to the baseline processor.

**Size.** We can decrease the size of the UCS from 128KB all the way down to 64KB with almost no impact on energy or runtime. This result shows that UCS utilizes storage space very efficiently and potentially allows the design of UCS processors with significantly less storage than non-UCS processors without negative impact on performance.

**Latency.** The most important sensitivity result is that we can increase the latency of the UCS, including the crossbar latency, from 3 cycles to 4 cycles with little impact on performance. Even at 5 cycles, the UCS processor's performance is still as good as the baseline (while using far less energy).

**Banks.** Varying the number of banks has almost no impact on the results (not shown), since UCS contention is not a major issue, even for a single-banked UCS, given the observed µcache hit rates.

## 5. RELATED WORK

There are three main areas of related work. First, there has been research in multi-level storage structures that strive to make the common case fast by optimizing for locality. Beyond the well-known multi-level cache hierarchies, there has been work in multi-level register files [31, 35, 39, 8], branch predictors [29, 14], scheduling windows [6], and store queues [1]. The Virtual Context Architecture backs up a register file cache using memory [23]. Researchers have also developed multi-level reorder buffers that enable instructions to speculatively retire from a small, fast reorder buffer into a larger history buffer [28, 11]. Similarly, the WIB enables a small, fast scheduling window to appear larger by backing it up with a buffer that holds instructions dependent on a long-latency instruction [20]. Similar to our µcaches, the filter cache filters requests to the L1 and beyond [17] in order to save energy. Kursun et al. [19] use the multi-level approach for multiple structures, including the caches, register file, and branch predictor.

Second, there has been considerable work in resizing structures in order to save energy, including resizable caches [3, 37, 38, 5] and resizable core structures, such as reorder buffers and scheduling windows [13, 25, 7, 2, 10]. This research is similar to ours in that it dynamically allocates storage based on need, but it differs in that it does not allow sharing across structures.

Third, there is a body of work that explores how to make caches appear to be more set-associative without slowing down their access times or increasing their energy consumption. This work is related to how we use the UCS to provide logically larger structures than the small µcaches. Way prediction [26] enables fast access and low energy consumption, if the prediction is correct. The Indirect Index Cache [12] and the V-Way Cache [27] split the tags from the data to achieve the same goal. Our L1 µcache designs are inspired by this work, but they are significantly modified to work well in the context of UCS.

## 6. CONCLUSIONS

We have developed the first microprocessor that can dynamically allocate on-chip storage among a set of storage structures. We replace each storage structure with a small, low-energy µcache, and we back up all of these µcaches with the UCS. By carefully managing the allocation and replacement of blocks, the µcaches can satisfy a large fraction of requests at very low time and energy cost. Misses in the µcache take longer than hits in the original structure, but this is compensated by faster hits in the µcaches. Because blocks can be placed anywhere in the UCS, space can be dynamically allocated to the most critical structure and is used more efficiently than in separate, set-associative structures. Results show an average energy savings of 36% and an average speedup of 9.5%. We conclude that the UCS approach is a viable alternative to entirely independent structures dominant today.

## 7. ACKNOWLEDGMENTS

This work is supported in part by the National Science Foundation under grant CCF-0444516, the National Aeronautics and Space Administration under grant NNG04GQ06G, Intel Corporation, and a Warren Faculty Scholarship. We thank Fred Bower, Alvy Lebeck, Tong Li, Anita Lungu, and Jaidev Patwardhan for their helpful feedback on this research.

## 8. REFERENCES

[1] H. Akkary, R. Rajwar, and S. T. Srinivasan. Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors. In *Proc. of 36th Annual Int'l Symposium on Microarchitecture*, Dec. 2003.

[2] D. H. Albonesi. Dynamic IPC/Clock Rate Optimization. In *Proc. of the 25th Annual International Symposium on Computer Architecture*, pages 282–292, June 1998.

[3] D. H. Albonesi. Selective Cache Ways: On-Demand Cache Resource Allocation. In *Proceedings of the 32nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 248–259, Nov. 1999.

[4] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *IEEE Computer*, 35(2):59–67, Feb. 2002.

[5] R. Balasubramonian et al. Memory Hierarchy Reconfiguration for Energy and Performance in General-Purpose Processor Architectures. In *Proc. of the 33rd Annual Int'l Symposium on Microarchitecture*, pages 245–257, Dec. 2000.

[6] E. Brekelbaum et al. Hierarchical Scheduling Windows. In *Proc. of the 35th Annual International Symposium on Microarchitecture*, pages 27–36, Nov. 2002.

[7] A. Buyuktosunoglu et al. A Circuit Level Implementation of an Adaptive Issue Queue for Power-Aware Microprocessors. In *Proc. of the 11th Great Lakes Symposium on VLSI*, pages 73–78, 2001.

[8] J.-L. Cruz et al. Multiple-Banked Register File Architectures. In *Proc. of 27th Annual Int'l Symposium on Computer Architecture*, pages 316–325, June 2000.

[9] S. Dutta, K. J. O'Connor, and A. Wolfe. High-Performance Crossbar Interconnect for a VLIW Video Signal Processor. In *IEEE International ASIC Conference*, pages 45–50, Sept. 1996.

[10] D. Folegnani and A. González. Energy-Effective Issue Logic. In *Proc. of 28th Annual International Symposium on Computer Architecture*, pages 230–239, July 2001.

[11] C. Gniady, B. Falsafi, and T. Vijaykumar. Is SC + ILP = RC? In *Proc. of the 26th Annual International Symposium on Computer Architecture*, pages 162–171, May 1999.

[12] E. G. Hallnor and S. K. Reinhardt. A Fully Associative Software-Managed Cache Design. In *Proc. of 27th Annual International Symposium on Computer Architecture*, pages 107–116, June 2000.

[13] A. Iyer and D. Marculescu. Power Aware Microarchitecture Resource Scaling. In *Proc. of Design, Automation and Test in Europe Conference*, Mar. 2001.

[14] D. J. Jimenez, S. W. Keckler, and C. Lin. The Impact of Delay on the Design of Branch Predictors. In *Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2000.

[15] N. P. Jouppi and S. J. Wilton. An Enhanced Access and Cycle Time Model for On-Chip Caches. DEC WRL Research Report 93/5, July 1994.

[16] S. Kaxiras, Z. Hu, and M. Martonosi. Cache Decay: Exploiting Generational Behavior to Reduce Cache Leakage Power. In *Proc. of the 28th Annual International Symposium on Computer Architecture*, July 2001.

[17] J. Kin, M. Gupta, and W. H. Mangione-Smith. The Filter Cache: An Energy Efficient Memory Structure. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 184–193, Dec. 1997.

[18] K. Krewel. Alpha EV7 Processor: A High-Performance Tradition Continues. *Microprocessor Report*, April 2005.

[19] E. Kursun et al. An Evaluation of Deeply Decoupled Cores. *Journal of Instruction-Level Parallelism*, 8:1–20, Apr. 2006.

[20] A. R. Lebeck et al. A Large, Fast Instruction Window for Tolerating Cache Misses. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 59–70, May 2002.

[21] L. Li et al. CCC: Crossbar Connected Caches for Reducing Energy Consumption of On-Chip Multiprocessors. In *DSD '03: Proc. of the Euromicro Symposium on Digital Systems Design*, page 41, 2003.

[22] S. Manne, A. Klauser, and D. Grunwald. Pipeline Gating: Speculation Control for Energy Reduction. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 132–141, June 1998.

[23] D. W. Oehmke et al. How to Fake 1000 Registers. In *Proc. of the 38th Annual International Symposium on Microarchitecture*, pages 7–18, Nov. 2005.

[24] S. Palacharla and J. E. Smith. Complexity-Effective Superscalar Processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 206–218, June 1997.

[25] D. Ponomarev, G. Kucuk, and K. Ghose. Reducing Power Requirements of Instruction Scheduling Through Dynamic Allocation of Multiple Datapath Resources. In *Proc. of the 34rd Annual International Symposium on Microarchitecture*, pages 90–101, Dec. 2001.

[26] M. Powell et al. Reducing Set-Associative Cache Energy via Way-Prediction and Selective Direct-Mapping. In *Proceedings of the 34rd Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2001.

[27] M. K. Qureshi, D. Thompson, and Y. N. Patt. The V-Way Cache: Demand-Based Associativity via Global Replacement. In *Proc. of 32nd Annual Int'l Symposium on Computer Architecture*, pages 544–555, June 2005.

[28] P. Ranganathan, V. S. Pai, and S. V. Adve. Using Speculative Retirement and Larger Instruction Windows to Narrow the Performance Gap between Memory Consistency Models. In *Proc. of the Ninth ACM Symposium on Parallel Algorithms and Architectures*, pages 199–210, June 1997.

[29] G. Reinman, T. Austin, and B. Calder. A Scalable Front-End Architecture for Fast Instruction Delivery. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 234–245, May 1999.

[30] J. A. Rivers et al. On High-Bandwidth Data Cache Design for Multi-Issue Processors. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 46–56, Dec. 1997.

[31] R. M. Russell. The Cray-1 Computer System. *Communications of the ACM*, 21(1):63–72, Jan. 1978.

[32] T. Sherwood et al. Automatically Characterizing Large Scale Program Behavior. In *Proc. of the Tenth Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.

[33] K. Skadron et al. Temperature-aware microarchitecture. In *Proc. of the 30th Annual International Symposium on Computer Architecture*, pages 2–13, June 2003.

[34] G. S. Sohi and S. Vajapeyam. Instruction Issue Logic for High-Performance Interruptable Pipelined Processors. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pages 27–34, June 1987.

[35] J. A. Swensen and Y. N. Patt. Hierarchical Registers for Scientific Computers. In *Proc. of the 1988 Int'l Conf. on Supercomputing*, pages 346–354, July 1988.

[36] H. Wang, L.-S. Peh, and S. Malik. A Technology-Aware and Energy-Oriented Topology Exploration for On-Chip Networks. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1238–1243, 2005.

[37] S.-H. Yang et al. An Integrated Circuit/Architecture Approach to Reducing Leakage in Deep Submicron High-Performance Caches. In *Proc. of the Seventh IEEE Symposium on High-Performance Computer Architecture*, pages 147–157, Jan. 2001.

[38] S.-H. Yang et al. Exploiting Choice in Resizable Cache Design to Optimize Deep-Submicron Processor Energy-Delay. In *Proceedings of the Eighth IEEE Symposium on High-Performance Computer Architecture*, Jan. 2002.

[39] J. Zalamea et al. Two-Level Hierarchical Register File Organization for VLIW Processors. In *Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 137–146, Dec. 2000.