Design Space Optimization of Embedded Memory Systems via Data Remapping

RODRIC M. RABBAH, KRISHNA V. PALEM
VINCENT J. MOONEY III, PINAR KORKMAZ and KIRAN PUTTASWAMY
Georgia Institute of Technology

In this paper, we provide a novel compile-time *data remapping* algorithm that runs in linear time. This remapping algorithm is the first fully automatic approach applicable to pointer-intensive dynamic applications. We show that data remapping can be used to significantly reduce the *energy consumed* as well as the *memory size* needed to meet a user-specified performance goal (i.e., execution time) – relative to the same application executing without being remapped. These twin advantages afforded by a remapped program – reduced cache size and energy needs – constitute a key step in a framework for design space exploration: for any given performance goal, remapping allows the user to reduce the primary and secondary cache size by 50%, yielding a concomitant energy savings of 57%. Additionally, viewed as a compiler optimization for a fixed processor, we show that remapping improves the energy consumed by the cache subsystem by 25%. All of the above savings are in the context of the cache subsystem in isolation. We also show that remapping yields an average 20% energy saving for an ARM-like processor and cache subsystem. All of our improvements are achieved in the context of DIS, OLDEN and SPEC2000 pointer-centric benchmarks.

Categories and Subject Descriptors: B.3 [**Hardware**]: Memory Structures; D.2 [**Software**]: Software Engineering; D.2.2 [**Software Engineering**]: Design Tools and Techniques

General Terms: Algorithms, measurements, performance, design

Additional Key Words and Phrases: Design space exploration, power aware, data remapping

1. INTRODUCTION

In the embedded systems domain, the memory is not only a valuable resource in terms of its available size, but also a significant *power* or *energy* sink, often consuming as much as 45% of the total chip power [19]. Architects have primarily relied on hardware innovations to reduce the memory needs of a program, particularly since lower memory requirements translate to lower commercial costs. The latter is of particular importance in the embedded domain where profit margins are usually quite low. More significantly however, the impact on the system power and energy needs is often dramatic. While the cache component of the memory hierarchy offers a significant opportunity for optimizing a program's memory needs [6], it poses a significant challenge. Traditionally, compiler optimizations have played an important role in improving the static memory footprint of a program [17, 18, 20]. These include various control and data transformations for achieving some of the same goals [4, 10, 15].

The focus of this paper is on a compile-time *data reorganization* or *remapping* transformation [22] that achieves the above goals in the context of a processor with two levels of cache. *We are able to show that our technique allows a program to achieve the same overall running time with just half the cache resources, when compared to a program that has not been reorganized.* Intuitively, and as detailed in the sequel, when the cache memory is halved, the corresponding power and energy requirements are also halved.

Rodric M. Rabbah, Krishna V. Palem, Vincent J. Mooney III, Pinar Korkmaz and Kiran Puttaswamy
Center for Research on Embedded Systems and Technology
School of Electrical and Computer Engineering
Georgia Institute of Technology
777 Atlantic Drive, Atlanta, GA 30332
{rabbah, palem, mooney, korkmazp, kiranp}@ece.gatech.edu

This work is supported in part by DARPA contracts F33615-99-1499 and F30602-00-2-0564.

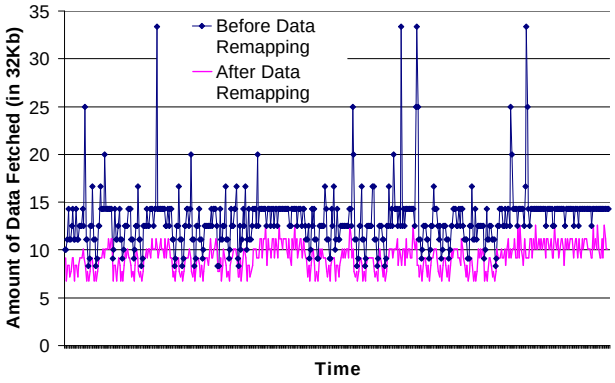


Figure 1. Amount of data fetched before and after data remapping.

The implications of this result are two-fold. First, our reorganization technique will allow embedded systems designers to pick a system whose cache memory and concomitant power needs are half of those achieved without our technique! In this sense, data reorganization can be viewed as an important component of a tool for *design-space exploration*, which can help lower the memory cost and power needs significantly, without compromising execution time. Second, when viewed as a conventional compiler optimization for a fixed target processor, it can be used to improve the performance of an application in the context of execution time as well as energy consumption. All of our work reported here is based on hardware models and an instruction set architecture (ISA) for the ARM family of processors, using floating point[1] and integer benchmarks.

Stated in simple terms, the proposed is an efficient remapping of the application's data layout in memory, such that data elements that are accessed contemporaneously are placed together in memory. Hence, remapping improves the spatial locality of data items that also share temporal locality. Specifically, in the absence of remapping, much of the data delivered to the cache is often needlessly fetched because of a lack of locality. In Figure 1, we plot the amount of data that is delivered to the cache for successive time slices throughout the execution of a representative benchmark. As a result of our remapping, the application's working-set size is effectively reduced – that is, the amount of data that is fetched is reduced by 30%. Consequently, the remapping transformation achieves its impressive improvement by ensuring that the ratio of the number of items found in cache (cache-hits) to those that are fetched from main memory (cache-misses) remain the same, with half the memory size, without compromising the application execution time.

Traditionally, data reorganization has been used to improve the execution time of applications for a fixed target processor [3, 9, 13, 18, 28]. To a large extent, previous work in this area has been a semi-automated process, and in the context of pointer-based programs – ubiquitous to the C programming language used extensively in the embedded systems domain – it has been mainly restricted to memory that is statically allocated. Our algorithm is (i) the first that is fully automated, (ii) applicable in the context of pointer-based programming languages with dynamic memory allocation support and (iii) is light weight with a running time linear in the size of the program.

In the next section, we will summarize the impact of data remapping on design space exploration. The remapping algorithm will be detailed in Section 3. Our evaluation framework and results follow in Sections 4 and 5 respectively. Section 6 summarizes and concludes the paper.

[1]Our simulation environment models an ARM-like processor but also includes floating point support.

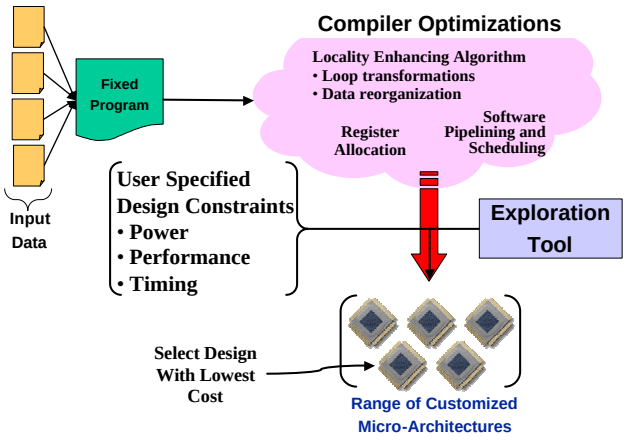


Figure 2. The concept of design space exploration.

Table 1. The impact of data remapping on power and cost during design space exploration.

Benchmark	Performance Goal (10^6 cycles)	Processor Type	Processor Cost	Before Remapping			After Remapping		
				L2 Size	L2 Cost	Total Cost	L2 Size	L2 Cost	Total Cost
179.ART	12,464	SMJ320C6701	$0.85	1MB	$19.38	$20.23	0MB	$0	$0.85
PERIMETER	516	SA 110	$66.03	2MB	$48.00	$114.03	1MB	$19.38	$85.41
TREEADD	877	SA 110	$66.03	1MB	$19.38	$85.41	.5MB	$17.80	$83.83

2. DATA REMAPPING AND DESIGN SPACE EXPLORATION

Broadly, our results and contributions fall into two categories. The first, and perhaps more important, category from the embedded systems perspective is in the context of design space exploration. In this sense, data remapping is an important step in exploring the design space when a system is assembled. This novel and interesting application of data remapping is self-evident and will be detailed next. However, data reorganization is often presented as a compiler optimization for a fixed target processor. To this end, the impact of data remapping in a more traditional role shall be discussed in Section 5.

The goal of design exploration, illustrated in Figure 2, is to fix the program under consideration and to vary its performance via optimizations, in search of the best hardware configuration. In our case, we shall focus on the cache subsystem and seek to optimize its energy and cost requirements. To this end, data remapping has proven to be a powerful tool. As shown in Table 1, remapping preserves the application's performance with half the cache size for three example benchmarks. Specifically, given a fixed execution time goal of 600 million cycles for the application PERIMETER, remapping allows us to use a 1MB secondary cache instead of 2MB for a total saving of $28.62 which is 25% of the cost[2]. More generally, we have demonstrated that this improvement is consistent in the context of several applications including floating-point and integer applications such as neural network simulation, large database management, image matching, and scientific computation from the Data Intensive Systems (DIS) [11], OLDEN [16] and SPEC2000 [23] benchmark suites. These results are summarized in Table 2. For each program, we show the reduction in overall energy consumed in the cache subsystem as well as the accompanying performance change where a negative number implies

[2]We use a StrongARM 110 processor from Intel for the TREEADD and PERIMETER integer benchmarks. We use a Texas Instruments processor for 179.ART, a floating point benchmark. The price quoted for the latter processor assumes that a quantity of at least 1,000 is purchased, while the price for the former is for a quantity of one. For the 2MB L2 cache, we use two 1MB Toshiba TC55W800FT-55 chips, each at a cost of $24. For the 1MB L2 cache, we use two 0.5MB Toshiba TC55V400AFT7, each at a cost of $9.19. For the 0.5MB L2 cache, we use four 128KB Cypress CY62128VL-70SC chips, each at a cost of $4.425.

Table 2. Halving power consumption through cache size reductions using remapping.

Benchmark	% L1+L2 Energy Reduction (J)	% Execution Cycles Reduction
164.GZIP	40.60	-0.66
179.ART	84.65	64.36
FIELD	38.05	-0.2
HEALTH	62.63	14.31
PERIMETER	58.45	22.80
TREEADD	58.56	10.46
TSP	57.05	21.77
Average	57.14	19.00

degradation, versus a positive number implying improvement. As seen in the last row of the table, the average energy improvement is 57.14%.

All of the energy estimates were derived from the well-known and widely used models due to Kamble and Ghose [12]. The memory behavior is measured using a processor simulator that supports an ARM ISA [7]. In Section 4 we will detail the structure, accuracy and other validation issues related to the evaluation framework. Our modeling of the processor power dissipation is based on an industry-standard power estimation tool and thus is fairly accurate [14].

3. DATA REMAPPING ALGORITHM

Locality enhancing optimizations amortize the cost of expensive memory accesses by improving data reuse and spatial locality. The latter is a notion of address adjacency in a memory reference stream. It is exploited in hardware by fetching a *set* of neighboring data items rather than delivering one element of the set at time. Since memory references are time consuming, loading and storing a set (cache block) of data items in a lower level of the hierarchy amortizes the cost of a memory access. However, if a reference stream does not exhibit address adjacency, valuable resources are wasted as data is unnecessarily fetched and cached. The proposed is a remapping of the elements into new sets, such that data items that are likely to be used together belong to the same cache block. Consequently, a greater percentage of the data that is fetched and stored in the cache will in fact be used (Figure 1). Furthermore, since the number of unnecessary data items that are cached is significantly reduced, the total required cache size is also reduced. As a result, remapping will often satisfy the design constraints with less hardware investments and hence, less energy dissipation (Section 2).

In order for the remapping to be effective – from a performance and especially power perspective – it must be achieved without actual data movement. Otherwise, the cost of online data relocation shadows any gains. To this end, our schema relies on a compiler coordinated placement of data, and introduces new offset computation functions to calculate the location of a data field relative to a known address. The remapping is customized such that the new layout exhibits (on average) a better correlation with the application reference sequence. This is in contrast to control optimizations where the access pattern is tailored to the memory layout. Such optimizations are often restricted in pointer-centric applications where data dependencies may not be readily resolved by a compiler.

3.1 Overview of Algorithm

The targets of our optimization are record data types ubiquitous to real-world, pointer-heavy applications. A record is a set of diverse data types grouped within a unique declaration; we shall refer to elements of the set as *fields* and instances of a record as *objects*. The specific focus on data records is self-evident. Consider for example a function that searches through a linked list of records and replaces a certain data item matching a search key. Each record consists of a *key* field, a *datum* field and a *next* field pointing to the next record in the list. Here, the *key* and *next* fields will be accessed in succession and more frequently (hot fields) than the datum field (cold field). Therefore, it would prove beneficial to fetch and cache as many hot fields as possible with each block access. To this end, a remapping strategy that collocates the

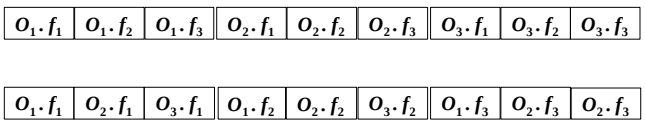


Figure 3. Two example memory access patterns ($|T| = 9$) for three objects O_1, O_2 and O_2 of record type R with fields f_1, f_2 and f_3. $O_j.f_k$ represents the k^{th} field of the j^{th} instance of a record R. For $B = 3$, $NAP(R)$ equals $\frac{7}{9}$ for (a) and 0 for (b).

key and *next* fields of various objects in the same cache block, and allocates all the of *datum* fields to a separate block, will improve the program spatial locality which in-turn favorably impacts memory system behavior. Note that packing field-pairs in the same block (i.e., a block containing *key* and *next* fields) does not offer an advantage over individual field packing (i.e., a block containing only *key* fields) since the same number of blocks will eventually be fetched from memory.

Our schema is an innovative combination of field reordering and customized placement such that the new data layouts exhibit better spatial locality. *The remapping optimization consists of three phases.*

(1) *Gathering Phase.* An analysis of the application memory access patterns is performed to identify record types that will benefit from remapping.

(2) *Remapping of Global Data Objects.* We first present the remapping strategy in the context of global data objects since they are often encountered in large applications. Next, we generalize the technique to dynamically allocated objects. We do not consider stack-allocated objects for remapping as they are often small and exhibit good locality.

(3) *Remapping of Dynamic Data Objects.* The key technical features of this work are geared toward pointer-centric applications and aim to preserve program semantics in the presence of pointer variables; a pointer variable is a variable whose value is the memory location (address) of another variable. Our optimization applies to programming languages such as *C* which associate physical meaning with the syntactic declaration of a record.

3.2 Gathering Phase

An arbitrary application of the remapping strategy to all data objects in a program does not necessarily increase spatial locality. Some data structures may not exhibit the requisite reference behavior to justify remapping. Although it is desirable to reorder data in memory to match all reference sequences, it is not computationally tractable [21]. To this extent, we analyze memory access patterns along program hot-spots [1] and select candidates for data remapping accordingly. The analysis is geared to characterize how well a traditional data layout is suited for various program memory access patterns or MAPs.

Consider the example memory access patterns shown in Figure 3 and let us assume a cache may accommodate three fields at a time, and that a block of the same size is used to deliver data from memory. In case (a), the reference pattern is such that the best data layout would assign the fields of object O_1 to one block, those of O_2 to another and similarly for O_3. This leads to a total of three cache misses, occurring on the access to $O_1.f_1$, $O_2.f_1$ and $O_3.f_1$. In case (b), the reference pattern warrants either an alternate layout or a larger cache. Otherwise, a total of nine misses will occur, one for each reference. That is, the access to $O_1.f_1$ will lead to the delivery of $O_1.f_2$ and $O_1.f_3$, which fills our cache. The next access however is to $O_2.f_1$ which will lead to a cache miss and the eviction of the currently cached data (and so one for the other references). In order to avoid redundant memory accesses, a larger cache is necessary, and in this case, one that is three times the current capacity. However, as noted earlier, larger caches incur greater investments. Hence, it is more desirable to modify the data layout such that data items in the block to be fetched are replaced with those that are more likely to be used.

Input: Program P, Cache Block Size B and Trace $T_R = (k, f)^*$ is a memory trace of all accesses to objects of record type R. The trace consists of a list of tuples (k, f), such that $T[i]$ for $0 < i$ represents the i^{th} tuple occurring in T, and it is an access to the f^{th} field of the k^{th} instance of record R.
Output: NAP for record type R occurring in Program P.

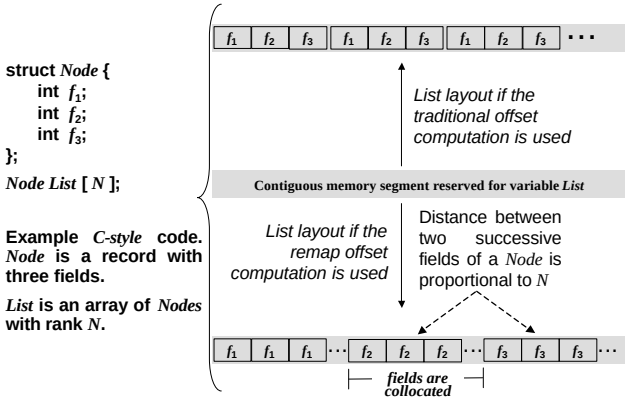
```
01. for j := B to |T| do
02.     for i := B − 1 downto 1 do
03.         (k_c, f_c) ← T[j]
04.         (k_p, f_p) ← T[j − i]
05.         if (k_c ≠ k_p) then
06.             if f_c and f_p belong to the same physical
                memory block then increment NAP(R)
07.         end if
08.     end for
09. end for
10. NAP(R) ← NAP(R) / (B(|T| − B))
```

Figure 4. Algorithm to compute the NAP for records in a program.

Although what is described is a pathological example, it illustrates the need for a proper characterization of the mismatch between traditional data layouts and the application memory access patterns. Our proposed analysis characterizes the mismatch as the *neighbor affinity probability* or NAP. The measured value may range from zero to one, where the latter indicates that the data layout is well suited for the analyzed reference pattern (i.e., high probability of a block containing successive data accesses). The other extreme indicates that the data layout does not exhibit any correlation to the memory access pattern (i.e., low probability of a block containing successive data accesses) and strongly warrants an alternate arrangement. The algorithm for computing the neighbor affinity probability is shown in Figure 4. For a fixed cache block size B, it analyzes an object reference trace (T) and computes the NAP for record types encountered in the program with an $O(|T|)$ running time. The block size enables a window-based analysis that searches for any occurrence of an access pattern resembling the one shown in Figure 3(a). Specifically, field references to different objects of the same record type are counted, but only if the fields reside in the same physical memory block (line 6). If this does not occur often enough, then it may prove worthwhile to collocate fields of different objects to increase the likelihood of a cache hit. If, on the other hand, the MAPs indicate that successive memory references already reside in the same cache block, then the traditional data layout is effective and may not benefit from a reorganization. The analysis need not distinguish between access patterns matching the desired template (i.e., the relative order of fields in an access pattern is not captured). This is in contrast to previous work where the temporal behavior of data fields is tracked [8]. We have found that although the latter may supplement our analysis, it leads to marginal enhancements. Finally, the computed NAP is normalized (line 10) and may be combined with affinity information gathered using a different memory profile. Data types with an affinity lower than some threshold are *marked* for remapping. All other record types are left unaltered and are organized using *traditional* memory layout strategies specified by the programming language.

3.3 Remapping of Global Data Objects

Once candidate record types have been identified, the global program variables are filtered to isolate arrays of records. Each such object is traditionally allocated in a contiguous memory segment (*cluster*) with a statically known starting location (*base*) and size (*rank*). The location of a field within a cluster is computed using an *offset computation function* (OCF) which determines the offset to the target relative to the base. For example, consider a record with fields f_1, f_2, f_3,



(a) Layout of fields in a reserved memory segment using the *traditional* and *remapped* layout strategies for a sample source code.

Input: NAP-annotated Program *P*.
Output: Data remapped *P*.

01. for each global variable *V* in *P* do
02. if *V* is an array of records *R* then
03. if *R* is marked for remapping then
04. Attribute *V* with *GDRemap*
05. else
06. Attribute *V* with *GDNomap*
07. end if
08. end if
09. end for
10. perform code generation

(b) Algorithm for remapping global data objects.

Figure 5. Schema for remapping global data objects.

and a cluster of such records with a rank of one. The offset to field f_1 relative to the base is zero – the base location of the cluster is the same as the location of field f_1 for the first record of the array. The offset to field f_2 is equal to the size of field f_1. Similarly, the offset to field f_3 is equal to the size of fields f_1 and f_2. This can be generalized to clusters of any rank as shown in Equation 1.

In order to improve spatial locality within a cluster, our data remapping strategy manipulates the offset computation function to yield a desired object and field layout. To this end, we introduce the *remap offset computation function* shown in Equation 2 and illustrate the data layouts that result from the traditional and remap offset expressions in Figure 5(a).

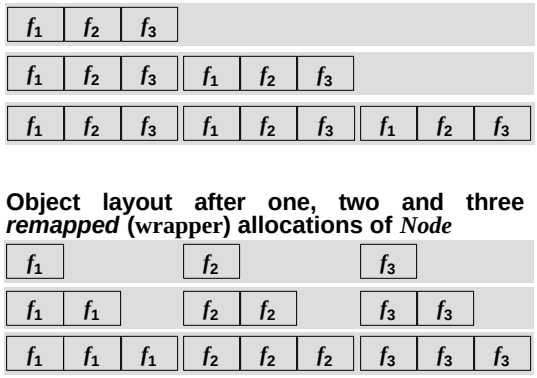
The remapping transformation is desirable for record types with low NAP, as the respective fields of various objects in the cluster are now adjacent – that is, the remapped data layout correlates well with the reference patterns shown in Figure 3b. In effect, if the NAP for a record type is low, then it follows that successive data references will likely not access fields of the same object.

The algorithm for remapping global data arrays is outlined in Figure 5(b). First, we attribute arrays of records in a program with either the traditional or remap offset expressions. Subsequently, during code generation, the associated expression is evaluated to compute the memory location of a referenced data item. Since the remapping is completely automated and performed by the compiler, expensive data relocation at run-time is not necessary. The two offset computation functions used for the purposes of this paper are

$$GDNomap(R_k.f) \;=\; (k-1) \times RecordSize(R) + \sum_{i=1}^{f-1} FieldSize(R.i) \tag{1}$$

$$GDRemap(R_k.f) \;=\; (k-1) \times FieldSize(R.f) + N \times \sum_{i=1}^{f-1} FieldSize(R.i) \tag{2}$$

where $R_k.f$ represents the f^{th} field of the k^{th} instance of a record R. We define $FieldSize(R.f)$ as the number of consecutive addressable units required to store field f, and $RecordSize(R)$ as the sum of $FieldSize(R.f)$ for all fields f in a record R.



(a) Layout of fields in a reserved memory segment for the traditional and remapped dynamic layout strategies.

Input: NAP-annotated Program P.
Output: Data remapped P.

```
01. for each statement S in P do
02.     if S is of the form x ← Allocate(RecordSize(R)) then
03.         if R is marked for remapping then
04.             replace S with x ← Wrapper(R)
05.             generate Wrapper(R) if necessary
06.         end if
07.     end if
08. end for
09. for each recordtype R in P do
10.     if R is marked for remapping then
11.         reorder the fields of R such that the most frequently
            used field has field index equal to 1
12.     end if
13. end for
14. perform code generation
```

(b) Algorithm for remapping dynamic data objects.

Figure 6. Schema for remapping dynamic data objects.

The essential difference between the two OCF is the last term. The latter staggers any two fields of a single object by a distance proportional to the size (N) of the cluster – we shall refer to N as the *stagger constant* and the term as a whole shall be called the *stagger distance*. However, since the remapping strategy is strictly applied to global data objects, the rank is readily available to the compiler and hence the stagger distance is statically computed. Therefore, the traditional and remapping strategies contribute the same run-time overhead.

3.4 Remapping of Dynamic Data Objects

The need for cache-conscious data placement is ever more important as applications increasingly rely on dynamically allocated objects [3, 8]. It is common for traditional allocation strategies to ignore the underlying memory hierarchy in favor of low run-time overhead. Unfortunately, such a scenario often results in poor interactions between data layout and program access patterns. Our methodology is to leverage the NAP analysis to identify suitable data types that would benefit from a controlled placement of newly allocated objects. The goal is to produce a field collocation layout as illustrated in Figure 6(a) and introduced previously for arrays of records. To this end, we use automatically generated light-weight *wrappers* around traditional memory allocation requests in the program – much like customized memory management mechanisms used in many applications where a large memory pool is allocated, and smaller portions within the pool are reassigned with successive allocation requests. However, unlike traditional custom memory management modules which tend to be complex [3, 9, 28] the generated wrappers are simple and efficient. Furthermore, the innovative combination of a custom memory allocator and new OCF allows for fine-grain control of field placement as opposed to object placement alone.

The algorithm for remapping dynamic data objects is shown in Figure 6(b). The first steps of the algorithm intercepts memory allocation requests and substitute custom allocation requests viz. a wrapper (lines 1-8). An example wrapper function is illustrated in Figure 7. The automatic generation of such wrappers is trivial and not discussed here. Note that the algorithm targets repeated single object allocations rather than dynamic allocations of arrays of record. The

Input: Record Type R and Stagger Constant N.
Output: Valid heap address where R is allocated.

/* Cluster, Base and Limit are persistent variables */
Initialize *Cluster*, *Base* and *Limit* to 0
if $Base = Limit$ then
    $Cluster \leftarrow$ reserve heap segment of size $N \times RecordSize(R)$
    $Base \leftarrow$ base address of *Cluster*
    $Limit \leftarrow Base + N \times FieldSize(R.f_1)$
end if
$Address \leftarrow Base$
$Base \leftarrow Base + FieldSize(R.f_1)$
return *Address*

Figure 7. An example wrapper function.

```
void Foo () {
    struct Node {
        int   f1;
        int   f2;
        int   f3;
    };
    Node List [ 100 ];
    Node *P;
    ...
    if ( select ) P = wrapper ( Node );
    else P = address of List [ k ];
    Print ( P® f2);
}
```

(a)

```
R1 = [ P ] + DDNomap (P® f2);

R2 = [ P ] + DDRemap (P® f2);
P0 = [ P ] > Stack Pointer Register
R1 = R2 if P0

R3 = Load R1

( note [ P ] represents the contents of P )
```

(b)

Figure 8. In (a) the value of *select* may not be statically known. P may alias a remapped record or a static record. The code generated for the expression $P \rightarrow f_2$ is shown in (b) and the dynamic disambiguation code is highlighted.

remapping strategy used for global arrays may be applied to dynamic ones. In this case however, the size of a dynamically allocated array may not be available to the compiler. Furthermore, an application may allocate several such arrays, each of a different size. To this extent, the optimization will incur some run-time overhead as the stagger distance is computed online. A special scenario arises when the compiler is able to determine that all dynamic arrays of a given record type are of the same size, or alternatively, that a suitable maximum size can be used. In such a case, the stagger distance is statically fixed and the wrapper adjusted accordingly.

Once all wrapper allocations are in place, the code generator calculates the field offset for a given pointer access. If it can be determined that a pointer aliases a dynamically allocated record, the compiler evaluates a remapping OCF expression (Equation 4). Similarly, the code generator uses the traditional offset calculation (Equation 3) for pointer variables that alias static records. When the compiler is unable to disambiguate a data alias, we evaluate both expressions and rely on a run-time comparison of the pointer value against the *stack pointer register* to determine the proper offset (Figure 8). This is possible since the remapping is restricted to heap objects and does not alter the layout of stack objects. We used a variation of Steensgaard points-to analysis [24] to statically disambiguate an application's pointer references. The simple run-time disambiguation, supplementing the compiler analysis, was found to be highly effective, contributing less than a 5% increase to the total dynamic instruction count of an application [22].

The offset computation expressions used for dynamically allocated objects are

$$DDNomap(P \rightarrow f) = \sum_{i=1}^{f-1} FieldSize(*P.i) \tag{3}$$

$$DDRemap(P \rightarrow f) = \sum_{i=1}^{f-1} StaggerConstant \times MaxFieldSize(*P) \tag{4}$$

where P is a pointer to a record of type $R = *P$ and $MaxFieldSize$ is the maximum $FieldSize$ of all fields f in a record R. The stagger constant is a compiler defined value that is equivalent to the rank of an array used earlier to remap global data objects. Note that a run-time disambiguation is not necessary for the first field of a record (i.e., when $f = 1$) since $DDNomap$ and $DDRemap$ evaluate to zero. Hence, the remapping algorithm modifies the record layout such that the most frequently used field has an index of one (lines 9-13 in Figure 6(b)).

Table 3. Execution time, power and energy results for PERIMETER.

Execution Cycles (10^6 of cycles)	Execution Time (ms) (100 Mhz Clock)	Power (W)	Energy (J)
0.53	53	0.99	0.052
1.29	129	0.97	0.125
6.17	617	0.96	0.592

Table 4. Execution time, power and energy results for TREEADD.

Execution Cycles (10^6 of cycles)	Execution Time (ms) (100 Mhz Clock)	Power (W)	Energy (J)
0.16	16	0.99	0.0158
0.56	56	0.99	0.0554
1.44	144	1.01	0.1454

4. EXPERIMENTAL METHODOLOGY

In the following, we shall detail the power models and simulation environments used to evaluate data remapping in a design space exploration context.

4.1 The Target Processor

To estimate power/energy consumption of an ARM-like processor core, we use a semi-custom VLSI design methodology. We obtained a Verilog model of an ARM-like processor core from the University of Michigan [25]. We synthesized the core using Synopsys Design Compiler targeted toward a TSMC 0.25μ library from LEDA Systems, Inc. We used a clock cycle time of 10ns (100MHz clock). The processor is a straightforward 5-stage RISC design able to execute the ARM ISA. The synthesized area of the processor core is approximately 250,000 NAND gate equivalents in the LEDA TSMC 0.25μ standard cell library. This approach to processor design, while not the industry standard, is becoming more common with companies like Tensilica whose product is a synthesizable processor core with a customizable ISA.

Given our synthesized core, we use the Synopsys Power Compiler to estimate power/energy consumption. Briefly, the Synopsys Power Compiler works as follows. First, we compile each benchmark to generate instruction and data in a format that the Verilog processor model can use to run the benchmark. Next, a Verilog simulation of the Register-Transfer Level (RTL) processor description in the Synopsys VCS simulator collects the switching activity (toggle rate) on each wire in the processor design. This simulation is very time-consuming (typically one million assembly instructions take three hours on a Sun Ultra 80 with four 450MHz Sparc processors and 4GB of memory). Finally, the switching activity is used together in conjunction with technology parameters to estimate dynamic and static power dissipation for the particular technology chosen (in our case, TSMC 0.25μ CMOS Technology). Thus, while typically used for ASIC design, we use the Synopsys Power Compiler to estimate power consumption of an ARM-like processor.

We show some results in Tables 3 and 4. Note that the power consumption is constant. This is likely due to the fact that in a simple RISC processor with one ALU, the datapath is always busy, and thus the power variation is minimal. More details about our modeling of the processor core are contained in a technical report [14].

4.2 The Compilation Environment

Benchmarks from the DIS, OLDEN and SPEC2000 suites were selected for detailed analysis. The OLDEN benchmarks provide a common frame of reference with previous work on data reorganization [9, 13, 28]. The others provide insight into larger programs. The benchmarks were executed using large input sets, whereas profile information was gathered using much smaller workloads (e.g. a trace size of few million memory instructions sampled along program hot spots). Table 5 summarizes the benchmarks and input workloads used for our experiments.

A short description of each benchmark is as follows. 164.GZIP is an integer SPEC benchmark. It utilizes a dynamically allocated array of records during decompression. 179.ART is a floating point benchmark from the SPEC suite. It dynamically allocates an array of records at startup, which is heavily used throughout execution. FIELD is a benchmark from the DIS suite. It uses a statically allocated array of records that is repeatedly searched and modified at random. The remaining benchmarks are memory intensive and allocate substantial amounts of heap objects. The primary data structure used in HEALTH is a linked list to which elements are added and removed. PERIMETER and TREEADD respectively allocate quad and binary trees at program start-up and do not subsequently modify them. TSP creates a quad-tree at program startup that is repeatedly updated.

Table 5. Benchmarks, workloads and main memory footprints.

Name	Workload	Memory Footprint
164.GZIP	test	15Mb
179.ART	test	small
FIELD	11654 Tokens	small
HEALTH	8 Levels, 100 Units 100	41Mb
PERIMETER	11Kx11K	146Mb
TREEADD	22 Levels, 20 Iteration	64Mb
TSP	1M Cities	40Mb

The remapping algorithms were implemented in TRICEPS [27], a publicly available infrastructure for compiler research based on TRIMARAN. It provides a common and uniform platform for verification and validation of results. It also includes an ARM code generator, an ARM-like processor simulator, and a smart memory and cache hierarchy simulator (SMACHS). The algorithms were implemented in the compiler front-end, where type information is available. The benchmarks were compiled using classic and high-level optimizations which include loop unrolling, copy propagation, common subexpression elimination, dead code elimination, and aggressive register allocation. The ARM-like simulator was configured as a single issue processor with stall on-use semantics. Various memory primary and secondary cache organizations and bus width were used and are reported throughout the paper where appropriate. The memory hierarchy includes streaming support and uses read/write-allocate semantics.

4.3 Model of Cache Power Consumption

To model energy consumed by the primary and secondary caches, which we assume to be SRAM, we use the approach of Kamble and Ghose [12]. In order to use their analytical model, we collect run-time statistics such as hit/miss counts and the ratio of read/write requests. These numbers are generated using SMACHS. Together with cache organization parameters such as cache capacity, line size and tag size, the model derives memory signal transition counts. One drawback to the model is that it does not model I/O pads. A more important drawback is that the model only accounts for dynamic power dissipation, which makes the model inaccurate for upcoming smaller geometries (such as 0.09 μ technology) with relatively large static (leakage) power dissipation. However, for the 0.25μ technology which we assume in our examples, dynamic power dissipation is still approximately two orders of magnitude greater than static power dissipation [26] and thus the model is still valid.

The model requires capacitance parameters, such as the metal wire capacitances as well as the gate and drain capacitances of the transistors in different parts of an SRAM circuit. We followed the example of Wilton and Jouppi [29] in calculating these values. For our purposes, we used the TSMC 0.25μ technology parameters to simulate the corresponding components of an SRAM circuit in HSpice. Further details are contained in a technical report [14].

5. RESULTS

We highlight here two different kinds of results. First, we detail the energy savings in the memory subsystem due to remapping. Next, we demonstrate savings in an architecture consisting of an ARM-like processor, an on-chip primary cache, and an off-chip secondary cache. Earlier, we summarized a third kind of a result. Namely, Table 1 of Section 1 illustrates how data remapping may be used to enable the substitution of less expensive COTS components during embedded system assembly.

5.1 Memory Subsystem Savings

Tables 6, 7, 8 and 9 show a subset of the L1 and L2 cache design space we explored for the benchmarks. Table 6 summarizes the energy and performance results for our baseline cache configuration, prior to remapping. Table 7 demonstrates the impact of remapping with respect to energy and performance using the original hardware configuration. In Table 8, we report the energy and performance results after remapping but for a secondary cache of half the original capacity. Fi-

Table 6. Execution time and energy results before remapping.
(L1=32KB, L1 line size=16 bytes, L2=1MB, L2 line size=32 bytes)

Benchmark	Execution Cycles	L1 Cache Energy (J)	L2 Cache Energy (J)	L1+L2 Energy (J)
164.GZIP	**1106079932**	**0.0311**	**0.085**	**0.116**
179.ART	**704713706**	**0.465**	**9.437**	**9.938**
FIELD	**1047393960**	**3.838**	**0.922**	**4.76**
HEALTH	**2616712073**	**1.293**	**21.489**	**22.782**
PERIMETER	**813958394**	**1.4189**	**7.151**	**8.5699**
TREEADD	**877485849**	**1.243**	**4.497**	**5.731**
TSP	**1077624556**	**1.868**	**8.676**	**10.544**

Table 7. Execution time and energy results after remapping.
(L1=32KB, L1 line size=16 bytes, L2=1MB, L2 line size=32 bytes)

Benchmark	Execution Cycles	L1 Cache Energy (J)	L2 Cache Energy (J)	L1+L2 Energy (J)	% Reduction in E from Table 6	% Reduction in Execution Cycles from Table 6
164.GZIP	**1083997353**	**0.0312**	**0.0848**	**0.116**	**0.000**	**2.00**
179.ART	**216812141**	**0.17**	**2.702**	**2.873**	**71.090**	**69.23**
FIELD	**1047626423**	**3.838**	**0.9219**	**4.7602**	**-0.004**	**0.00**
HEALTH	**2044289648**	**1.263**	**14.81**	**16.072**	**29.337**	**21.88**
PERIMETER	**628221147**	**1.4178**	**5.139**	**6.557**	**23.488**	**22.82**
TREEADD	**785358662**	**1.0344**	**3.3129**	**4.347**	**24.149**	**10.50**
TSP	**926038088**	**2.245**	**4.985**	**7.23**	**31.430**	**14.07**
Average	–	–	–	–	**25.640**	**20.07**

Table 8. Execution time and energy results after remapping.
(L1=32KB, L1 line size=16 bytes, L2=512KB, L2 line size=32 bytes)

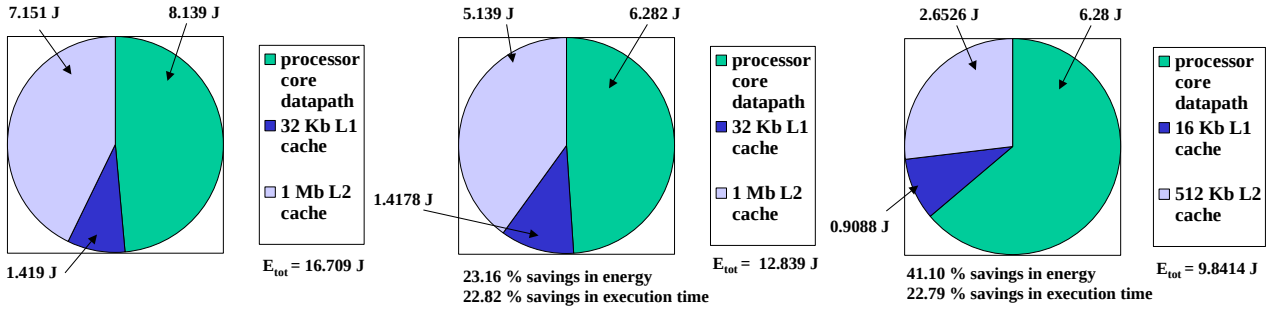
Benchmark	Execution Cycles	L1 Cache Energy (J)	L2 Cache Energy (J)	L1+L2 Energy (J)	% Reduction in E from Table 6	% Reduction in Execution Cycles from Table 6
164.GZIP	**1083997353**	**0.0312**	**0.0439**	**0.0751**	**35.258**	**2.00**
179.ART	**250995040**	**0.17**	**1.418**	**1.588**	**84.021**	**64.38**
FIELD	**1047626423**	**3.838**	**0.475**	**4.313**	**9.391**	**-0.02**
HEALTH	**2044289648**	**1.26**	**7.625**	**8.888**	**60.987**	**21.88**
PERIMETER	**628311657**	**1.4178**	**2.6489**	**4.0667**	**52.547**	**22.81**
TREEADD	**785407236**	**1.0344**	**1.7059**	**2.74**	**52.189**	**10.49**
TSP	**956363728**	**2.245**	**2.6103**	**4.855**	**53.955**	**11.25**
Average	–	–	–	–	**49.764**	**18.97**

nally, in Table 9, we demonstrate the impact of remapping on energy and execution time when the primary and secondary caches are half their original size.

Some results worth highlighting are that the largest energy reduction always occurs in the smallest cache configuration (Table 9). The largest execution time reduction, however, most often occurs using the original cache configurations with remapping enabled (Table 7). Intuitively, this makes sense. Clearly, the larger the caches, the faster the benchmark completes. With half sized caches, on the other hand, the number of cache entries is halved, thus halving the capacitance seen on any particular bit line. Since power is proportional to the capacitance and voltage (CV^2), halving the former (C) halves the power consumed. Also note that after remapping, the primary cache energy tends to increase. This is due to more L1 cache hits. However, the overall energy of both L1 and L2 is significantly reduced, due to the greatly decreased number of cache lines exchanged between L1 and L2.

Table 9. Execution time and energy results after remapping.
(L1=16KB, L1 line size=16 bytes, L2=512KB, L2 line size=32 bytes)

Benchmark	Execution Cycles	L1 Cache Energy (J)	L2 Cache Energy (J)	L1+L2 Energy (J)	% Reduction in E from Table 6	% Reduction in Execution Cycles from Table 6
164.GZIP	**1113402555**	**0.02**	**0.0488**	**0.0689**	**40.603**	**-0.66**
179.ART	**251159762**	**0.1078**	**1.4177**	**1.525**	**84.654**	**64.34**
FIELD	**1047626417**	**2.474**	**0.474**	**2.949**	**38.046**	**-0.02**
HEALTH	**2046953548**	**0.8014**	**7.7112**	**8.513**	**62.633**	**21.77**
PERIMETER	**628440207**	**0.9088**	**2.6526**	**3.561**	**58.448**	**22.80**
TREEADD	**785670341**	**0.6623**	**1.713**	**2.375**	**58.559**	**10.46**
TSP	**975110313**	**1.446**	**3.083**	**4.529**	**57.046**	**14.31**
Average	–	–	–	–	**57.141**	**19.00**



(a) Baseline energy consumption before remapping.

(b) Energy savings after remapping.

(c) Energy savings after remapping with smaller caches.

Figure 9. Design-space exploration for PERIMETER.

5.2 Energy Savings for an ARM-like Core

Whereas the previous section detailed the energy and related improvements achieved by data remapping in the context of the cache subsystem in isolation, it is important to understand its impact in the context of the entire microprocessor. To this end, we calculate the energy consumed by our ARM-like processor using the power measurement approach described in Section 4.1. We take the average power (J/s) of the largest simulation (e.g., 6.17 million cycles for PERIMETER, see Table 3) and multiply the average power by the execution time (J/s $\times$ s = J) for our 100MHz ARM-like model. Note that the average power consumed in the core varied very little (less than 3%), so we believe that the extrapolation is fairly accurate.

First, we show that remapping speeds up the overall application such that its completion time is faster on the target microprocessor. Therefore, by executing faster, the energy consumed by the processor after remapping is lower. This is illustrated in Figure 9(b) for PERIMETER, where the reduction in execution cycles is 22.82% relative to the baseline shown in Figure 9(a). The accompanying improvement in the energy consumed is 23.16%. The improvement is achieved by keeping the cache size fixed. *Second*, if we allow the cache size to change – in particular halving the L1 and L2 caches in the ARM-like processor – we show in Figure 9(c) that an energy savings of 41.10% can be achieved over the same baseline (Figure 9(a)). Note that, for the same reason noted in the previous section, the lowest energy configuration is not the same as the lowest execution time configuration.

6. RELATED WORK AND REMARKS

In the context of design space exploration, there has been excellent work reported for custom memory management and optimizations. For example, Catthoor et al. [5] perform an extensive exploration of memory organization for embedded system design, with an emphasis on storage and bandwidth optimization. In addition, Panda et al. [17] recently published a thorough survey of data and memory optimization techniques for embedded systems. Notably, researchers [18, 2] have explored a coordinated data and computation reordering for array-based data structures in Multi-Media applications. Our optimization extends current state-of-the-art design space exploration and custom memory management methodology to pointer-centric applications ubiquitous in embedded systems.

Other related work in data reorganization [8, 13] propose automated field-ordering algorithms that assign temporally related fields of a record to adjacent memory locations. The optimizations however offer only partial solutions, as they do not consider the interaction of fields among various instances of a record. Chilimbi et al. [9] and Truong et al. [28] described a data placement scheme to specifically address this issue. However, the proposed strategies are focused on optimizing performance with respect to execution time. Furthermore, the optimizations are (i) not completely transparent to the programmer, (ii) require some manual re-tooling of the application, (iii) incur significant run-time overhead as objects are dynamically relocated in memory, and (iv) may violate program correctness. By contrast, our approach is (i) targeted to reduce power and energy consumption in real systems, (ii) is completely automated, (iii) does not perform any run-time data movements, (iv) and preserves correctness for a much larger scope of applications. The ability of data remapping to significantly enhance locality without run-time data movements is key, especially from an embedded systems perspective. The cost of dynamic data relocation is often prohibitive from a performance point of view, and certainly from a power perspective.

There has been much previous work in power modeling of processors and memory. Generally, the reported techniques for processor modeling trade off slightly reduced accuracy (compared to gate-level simulation) for significantly faster simulation speed. Our simulation methodology falls in the more accurate but slow category. However, our models will need to evolve to account for leakage power and the plethora of low-power approaches to memory design.

REFERENCES

[1] T. Ball and J. Larus. Efficient path profiling. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, Dec. 1996.

[2] H. M. C. Kulkarni, F. Catthoor. Advanced data layout organization for multi-media applications. In *Proceedings of the Workshop on Parallel and Distributed Computing in Image Processing, Video Processing, and Multimedia*, May 2000.

[3] B. Calder, C. Krintz, S. John, and T. Austin. Cache-conscious data placement. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 139–149, Oct. 1998.

[4] J. Carter, W. Hsieh, M. Swanson, L. Zhang, A. Davis, M. Parker, L. Schaelicke, L. Stoller, and T. Tateyama. Memory system support for irregular applications. In *Workshop on Languages, Compilers, and Runtime Systems for Scalable Computers*, May 1998.

[5] F. Catthoor, S. Wuytack, E. DeGreef, F. Balasa, L. Nachtergaele, and A. Vandecappelle. *Custom Memory Management Methodology. Exploration of Memory Organization for Embedded Multimedia System Design*. Kluwer Academic Publishers, 1998.

[6] L. Chakrapani, P. Korkmaz, V. Mooney, K. Palem, K. Puttaswamy, and W. Wong. The emerging power crisis in embedded processors: What can a poor compiler do? In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 176–180, Nov. 2001.

[7] L. Chakrapani, K. Palem, and W. Wong. TRICEPS: Enhancing the TRIMARAN compiler infrastructure for StrongARM code generation. Technical Report CREST-TR-01-001, Georgia Institute of Technology, May 2001.

[8] T. Chilimbi, B. Davidson, and J. Larus. Cache-conscious structure definition. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–24, May 1999.

[9] T. Chilimbi, M. Hill, and J. Larus. Cache-conscious structure layout. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, May 1999.

[10] C. Ding and K. Kennedy. Improving cache performance of dynamic applications with computation and data layout transformations. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 229–241, May 1999.

[11] DATA-INTENSIVE SYSTEMS benchmark suite. www.aaec.com/projectweb/dis/.

[12] M. Kamble and K. Ghose. Analytical energy dissipation models for low power caches. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 143–148, Aug. 1997.

[13] T. Kistler and M. Franz. Automated data-member layout of heap objects to improve memory-hierarchy performance. *ACM Transactions on Programming Languages and Systems*, 22(3):490–505, May 2000.

[14] P. Korkmaz, K. Puttaswamy, and V. Mooney. Energy modeling of a processor core using synopsys and of the memory hierarchy using the kamble and ghose model. Technical Report CREST-TR-02-002, Georgia Institute of Technology, Feb. 2002.

[15] M. Lam, E. Rothberg, and M. Wolf. The cache performance of blocked algorithms. In *Proceedings of the Fourth International Conference in Architectural Support for Programming Languages and Operations Systems*, pages 63–74, Apr. 1991.

[16] OLDEN benchmark suite. www.cs.princeton.edu/ mcc/olden.html.

[17] P. Panda, F. Catthoor, N. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkrani, A. Vandercappelle, and P. Kjeldsberg. Data and memory optimization techniques for embedded systems. *ACM Transactions on Design Automation of Electronic Systems*, 6(2):149–206, Apr. 2001.

[18] P. Panda, N. Dutt, and A. Nicolau. Memory data organization for improved cache performance in embedded processor applications. *ACM Transactions on Design Automation of Electronic Systems*, 2(4):384–409, 1997.

[19] P. Panda, N. Dutt, and A. Nicolau. *Memory Issues In Embedded Systems-On-Chip, Optimizations and Exploration*. Kluwer Academic Publishers, 1999.

[20] P. Panda, N. Dutt, and A. Nicolau. On-chip vs. off-chip memory: The data partitioning problem in embedded processor-based systems. *ACM Transactions on Design Automation of Electronic Systems*, 5(3):682–704, July 2000.

[21] E. Petrank and D. Rawitz. The hardness of cache conscious data placement. In *Proceedings of the 29th Annual ACM Symposium on Principles of Programming Languages*, Jan. 2002.

[22] R. Rabbah and K. Palem. Data remapping for design space optimization of embedded cache systems. Technical Report GIT-CC-02-10, Georgia Institute of Technology, Mar. 2002.

[23] STANDARD PERFORMANCE EVALUATION CORPORATION CPU2000 benchmark suite. www.spec.org.

[24] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 32–41, Jan. 1996.

[25] The SimpleScalar-ARM power modeling project. www.eecs.umich.edu/˜jringenb/power/.

[26] S. Thompson, P. Packan, and M. Bohr. Mos scaling: Transistor challenges for the 21st century. Technical Report Q3, Intel Technology Journal, July 1994.

[27] TRICEPS: A TRIMARAN-based ARM code generator. www.trimaran.org/triceps.shtml.

[28] D. Truong, F. Bodin, and A. Seznec. Improving cache behavior of dynamically allocated data structures. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 322–329, Oct. 1998.

[29] S. Wilton and N. Jouppi. An enhanced access and cycle time model for on-chip caches. Technical Report 93.5, WRL Research Report, July 1994.