Heterogeneous Memory Management for Embedded Systems

Oren Avissar
ECE department
University of Maryland
College Park, MD 20742, U.S.A
oavissar@eng.umd.edu

Rajeev Barua
ECE department
University of Maryland
College Park, MD 20742, U.S.A
barua@eng.umd.edu

Dave Stewart
Embedded Research Solutions, LLC
9687F Gerwig Lane
Columbia, MD 21046
dstewart@embedded-zone.com

ABSTRACT

This paper presents a technique for the efficient compiler management of software-exposed heterogeneous memory. In many lower-end embedded chips, often used in micro-controllers and DSP processors, heterogeneous memory units such as scratch-pad SRAM, internal DRAM, external DRAM and ROM are visible directly to the software, without automatic management by a hardware caching mechanism. Instead the memory units are mapped to different portions of the address space. Caches are avoided because of their cost and power consumption, and because they make it difficult to guarantee real-time performance. For this important class of embedded chips, the allocation of data to different memory units to maximize performance is the responsibility of the software.

Current practice typically leaves it to the programmer to partition the data among the different memory units. We present a compiler strategy that automatically partitions the data among the memory units. We show that this strategy is optimal among all static partitions for global and stack data, and a good heuristic for heap data. For global and stack data, the scheme is provably equal to or better than any other compiler scheme or set of programmer annotations. Preliminary results show the benefits of optimal allocation: with just 20% of the data in SRAM, the formulation is able to decrease the runtime by 39% on average for our benchmarks vs. allocating all data to slow memory, without any programmer involvement. For some programs, less than 5% of data in SRAM achieves a similar speedup.

Keywords

Memory, heterogeneous, storage, embedded

1. INTRODUCTION

This paper presents a automatic compiler method for allocating program data among different heterogeneous memory units in embedded systems. The kind of embedded chips targeted are those without caches, and with at least two kinds of writable memory – only in such systems is intelligent memory allocation useful. Such embedded chips without caches and with multiple memory units are an important class of embedded chips, common in many lower-end embedded devices such as micro-controllers and some DSP chips. The writable memory units typically found in such systems include any two or more of: internal SRAM, external SRAM, integrated DRAM, external DRAM, and even EEPROM, which are writable, but with very high latency. In such chips, each of the memory units may have differing latencies and sizes, therefore the choice of memory allocation affects overall performance.

The allocation strategy presented in this paper models the problem using a 0/1 integer linear program, and solves it using commercially available software [8]. The formulation is provably optimal for global and stack data, and a good heuristic for heaps. It is easily adapted to both non pre-emptive and pre-emptive (context-switching) scenarios. For the first time ever, the solution automatically distributes the program stack among multiple memory banks, effectively growing the stack simultaneously in several places. Stack distribution is unusual – presently programmer annotations and compiler methods place the entire stack in one memory unit. The resulting flexibility allows a more custom allocation for better performance. Finally, the optimality guarantee ensures that the solution is equal to or better than any programmer-specified allocation using annotations, or any existing or future compiler method.

The method presented is motivated by a need to improve the quality of automatically compiled code. Compilers today, while better than before, still suffer from a large performance penalty compared to programs directly written in assembly language [3, 14]. This forces many performance-critical kernels to be written directly in assembly. Assembly programming has well-known disadvantages: more tedious, expensive and error-prone code development, difficulty in porting between different platforms, and a longer time-to-market between successive implementations [3, 14]. Further, optimizations that benefit from whole-program analysis, such as memory allocation, cannot be captured by rewriting certain kernels.

One of the major remaining impediments to efficient com-

CASES'01, November 16-17, 2001, Atlanta, Georgia, USA.

pilation is the presence of multiple, possibly heterogeneous memory units mapped to different portions of the address space. Many low-end embedded processors [11, 17, 10] have such memory units like on-chip scratch-pad SRAM, on-chip DRAM, off-chip DRAM, and ROM that are *not* members of a unified memory hierarchy. Caches are not used for reasons of real-time constraints, cost, and power dissipation. In contrast, the memory units in desktop processors are unified through caches. Directly addressed memory units in embedded processors require the software to allocate the data to different memories, a complex task for which good strategies are not available.

This work proposes a method for automatically allocating program data among the heterogeneous memory units in embedded processors without caches. The standard practice today is that the allocation is left to the programmer. An automated solution for a different kind of embedded processor has been proposed [13], namely, those for which the external memory has an internal hardware cache. We show, however, that the optimality criteria are very different without caches. Other related works are discussed later in the related work section.

Figure 1 outlines our method for heterogeneous memory management on embedded systems. The application program on the left is fed to our compiler analysis that derives the optimal allocation for the data. The compiler analysis incorporates application-specific runtime information through the use of profile data. The analysis is provided with the sizes and latencies of the memory units available on the target chip, as shown. The compiler analysis models the problem for global and stack data as a 0/1 integer linear programming problem and solves it using Matlab [8]. The solution is always provably optimal for handling global and stack data. A good heuristic is used for heap data. As depicted, the derived allocation specification is output to the linking stage of compilation. The linker adds assembly directives to its output code to implement the desired allocation. The resulting code not only improves performance but is likely to reduce energy consumption – it is well-known that compiler optimizations that reduce runtime usually reduce energy consumption as well [7].

This paper is organized as follows. Section 2 motivates the problem and outlines our approach. Section 4 shows a simple example to illustrate the tradeoffs involved. Section 5 describes our method for global variables. Sections 6 and 7 show how the formulation can be extended for stack variables and heap data, respectively. Section 8 presents some preliminary results. Section 3 describes related work; section 9 addresses certain real-world issues; while section 10 concludes.

2. MOTIVATION AND APPROACH

Multiple heterogeneous memory units in many embedded systems are motivated by the varying performance, cost and writability characteristics of different memory technologies. Typically, such chips contain some or all of the following: a small amount of on-chip SRAM, a moderate amount of on-chip DRAM, a moderate amount of off-chip SRAM, a large amount of off-chip DRAM, and some ROM to store programs and data constants. Each kind has its advantages. SRAM is fast but expensive, off-chip SRAM is somewhat slower, integrated on-chip DRAM is slower than SRAM but faster than external DRAM, while external DRAM is slow, but is the cheapest. ROM is cheap and non-volatile, but it cannot be written to. Certain kinds of ROM such as EEPROM are writable with high latency.

The organization of the memories is different from in desktop systems. While desktops also contain many of these memories, they automatically manage them hierarchically using caches. Caches, however, consume area and power, and make it difficult to provide real-time guarantees. Consequently, many embedded chips, except for some at the high end, do not use caches. Examples include the Motorola 68HC12 [10], Motorola MCore [11] and Texas Instruments TMS370Cx [17]. The lack of caches has meant that the different memories are mapped to different non-overlapping portions of the address space.

A result of the lack of caches is that the allocation of data to memories must be software-managed – in most systems today, it is left to the programmer. This work presents a compiler method to manage the data. Compiler methods are preferable to programmer directives as they do not require programmer effort; are portable across different systems; and are likely to make better decisions, especially for large, complex programs. The dominance of chips without caches in embedded systems implies that good compiler methods for data allocation will have a large impact.

Our profile-guided compiler method is static: it fixes the allocation at compile-time; data is not relocated to another location during runtime. Alternate strategies could be dynamic – software-managed caches [9, 5] are one class of dynamic algorithms. There are, however, no software caching strategies available today that are optimized for the class of embedded processors we target. Dynamic strategies are not studied in this work – future work might study such schemes. Significant challenges will need to be overcome in designing a software-caching scheme for embedded chips, including providing real-time guarantees in the face of unpredictable cache behavior; reducing software caching overhead; and restricting code size increase from the overhead instructions. Fortunately, profile data allows our static method to incorporate runtime information to some extent.

Even for static allocations, several factors make the problem a difficult one to solve optimally. Let us consider global, stack and heap variables. We present a formulation that is optimal for global variables among static partitions. Stack variables, however, unlike global variables, have limited lifetimes, allowing sharing of space between variables with disjoint lifetimes. This complicates the analysis, but we present a method that is able to retain the optimality guarantee for non-recursive procedures. For heap data, the situation is worse: no static method can be optimal for all heap data as the sizes and allocation frequencies are unknown for heap data. We describe a method that is a good heuristic.

3. RELATED WORK

For embedded processors with heterogeneous memory units that are not cache-managed, there has been little related work that automatically allocates data to banks while increasing performance. The usual approach is to leave the task to the programmer. Compiler methods are preferable to programmer directives for three reasons: they do not require programmer effort; are portable across different systems; and are likely to make better decisions, especially for large, complex programs.

To our knowledge, Panda et. al [13] and Sjodin et. al [16]

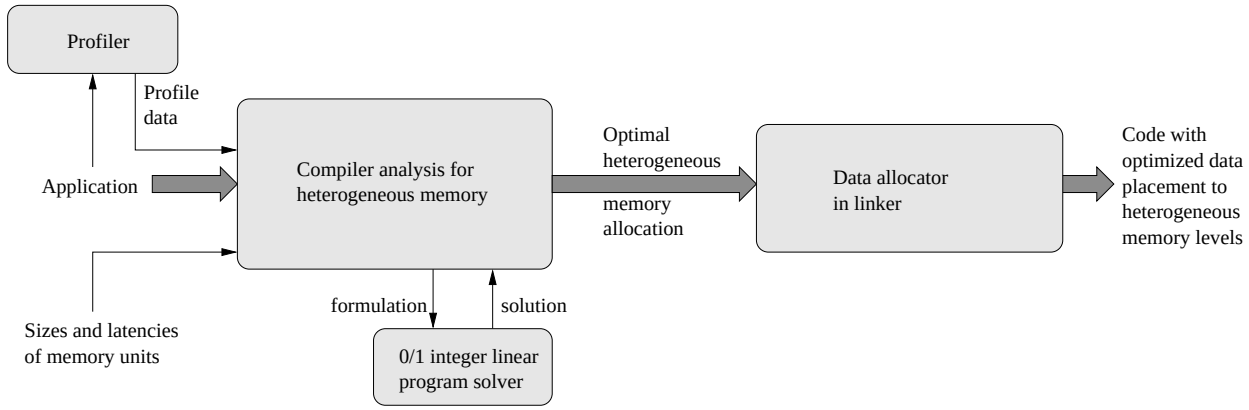

Figure 1: Flow diagram for method in paper. The thick arrows denote the primary flow of compilation and synthesis, while the thin arrows provide supporting data.

are the only published methods that aim to allocate data to on-chip and off-chip memories mapped to different portions of the address space. The architecture class targeted by Panda et. al [13] is different from ours, however: they target embedded processors that, in addition to having scratch-pad SRAM, use hardware-managed caches on top of slower, external memory[1]. The presence of caches completely changes the goals of the allocation strategy. Instead of aiming to reduce the number of accesses to data in external memory, it becomes far more important to ensure that those accesses hit in cache. Consequently, the goal of the method in [13] is to map the variables that are likely to cause the most conflicts to scratch-pad. It makes no attempt to maximize the number of accesses to scratch-pad, and thus is unsuitable for our architectural model.

The method proposed by Sjodin et. al [16] also differs from ours in several ways. Like our method, [16] also utilizes an allocation scheme that tries to keep variables with the highest number of accesses per byte in on-chip SRAM and allocate the less critical variables to slower external RAM. However, unlike our formulation [16] only addresses two memory levels (on-chip SRAM and external RAM) and does not offer any methods for extending to allocate stack or heap variables. Our formulation automatically handles N levels of memory with varying latencies and sizes. Also, we have extended formulations to account for both stack (local) and heap (malloc) variables. Another difference is that [16] primarily uses a static profiling scheme, which can be inaccurate – especially for larger programs. Our scheme always uses dynamic profiling which accounts for every load and store throughout the program.

As described earlier, our method yields a static allocation; dynamic strategies are possible. In a dynamic strategy, data may be moved from one location to another during program execution. One class of dynamic strategies are software-caching methods [9, 5] – these emulate a cache in fast memory using software. The tag, data and valid bits are all managed by compiler-inserted software at each program data access. Software overhead is incurred to manage these fields, though [9] compiler-optimizes away the overhead in some cases. [9] targets the primary cache, while [5] is intended for managing the secondary cache and assumes different costs – for this reason [9] is more applicable for the problem we consider.

[1] Our architectural model without caches is employed by a variety of embedded processors, as caches consume area and power, and make it difficult to provide real-time guarantees.

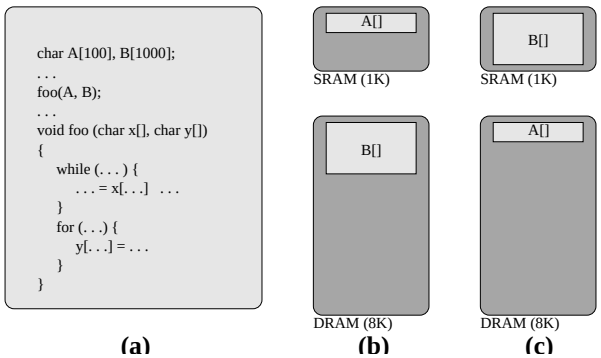

Figure 2: Example showing choices in heterogeneous memory allocation. (a) shows the source code of the application. (b) shows $A[]$ allocated to SRAM. Allocated portions are shaded lighter. (c) shows $B[]$ allocated to SRAM.

Dynamic strategies are not studied in this work – future work might study such schemes. Significant challenges will need to be overcome in designing a software-caching scheme for embedded chips, including providing real-time guarantees in the face of unpredictable cache behavior; reducing software caching overhead; and restricting code size increase from the overhead instructions. Fortunately, profile data allows our static method to incorporate runtime information to some extent.

4. EXAMPLE

Figure 2 shows a simple example that illustrates how a programmer or compiler might make decisions about data allocation. Figure 2(a) is the application program for which we wish to allocate the data. Two byte arrays $A[100]$ and $B[1000]$ are passed as arguments to procedure *foo* where they are accessed using formal arguments $x[]$ and $y[]$. Assume that the program is compiled for an embedded chip that has 1K bytes of fast scratch-pad SRAM, no on-chip

DRAM and 8K of slower external DRAM. The problem we are trying to solve is: which program variables should be allocated to which memory bank? It is clear that either array can individually fit in the 1K SRAM, but that both cannot simultaneously fit. For simplicity of illustration assume that no other accesses to A and B occur, although there is no such requirement in our method.

For the code in figure 2(a), the compiler needs to choose between the two possible data allocations shown in figures 2(b) and (c). In figure 2(b) $A[100]$ is in SRAM; in figure 2(c) $B[1000]$ is in SRAM. The choice between the two depends on access frequencies. Two cases illustrate the choice. In the first case, suppose the *while* loop at runtime actually executes for more iterations than the *for* loop, then the allocation in figure 2(b) is preferred as it makes more accesses to faster SRAM. In the second case, suppose the *for* loop executes for more iterations instead. In this case, the allocation in figure 2(c) is superior as it makes more SRAM accesses. Making estimates of relative access frequencies is difficult, however: many loops have bounds that are unknown at compile; further, data-dependent control flow makes static prediction difficult. Fortunately profile data gives good estimates provided the data set used is representative, and is far more accurate than static frequency prediction methods. Our allocation method uses profile data to find access frequencies of memory references.

The above example illustrates that for making good allocation decisions, the compiler must integrate at least three technologies. *First*, the general problem of optimal data allocation, which is NP-complete, must be solved either exactly or approximately using heuristics. We propose a method that is returns an optimal static solution using an integer programming framework. There is some evidence that 0/1 integer programming has fast solution times (under a minute on modern computers) even for large programs with a few thousand variables [1, 12]. Fortunately, the number of variables in our formulation is proportionate to the number of variables in the original program, which is usually no more than a few thousand for even large programs. Quick solution times are borne out by our results, where in all cases the solution was returned in under a minute. *Second*, the compiler must collect accurate frequency estimates using profiling, and use them to guide allocation. *Third*, in order to collect frequency counts for variables, the profiler must correlate accesses with the variables they access. For example, in figure 2(a), it must be known that $x[]$ is $A[]$ and $y[]$ is $B[]$. Knowing this statically requires computationally expensive inter-procedural pointer analysis. Moreover, pointer analysis may return ambiguous data if, for example, there is more than one call to *foo()* with different array arguments each time. For this reason, we avoid pointer analysis by taking a different approach – runtime address checks during profiling. During the profile run, each accessed address is checked against a table of address ranges for the different variables. A match indicates that the variable accessed has been found. This profile-based approach yields exact statistics, unlike the inexact information using pointer analysis.

5. FORMULATION FOR GLOBAL VARIABLES

Here we present the formulation for global variables; it is extended to handle stack and heap variables later. The following symbols are used[2]:

U = number of heterogeneous memory units;
T_{rj} = Time (latency) to read memory unit $j \in [1, U]$ in cycles;
T_{wj} = Time (latency) to write memory unit $j \in [1, U]$ in cycles;
M_j = Size of memory unit $j \in [1, U]$ in bytes;
G = number of global variables in application;
$v_i = i^{th}$ global variable, $i \in [1, G]$;
$N_r(v_i)$ = Number of times v_i is read (from profiling);
$N_w(v_i)$ = Number of times v_i is written (from profiling);
$S(v_i)$ = Size of variable v_i in bytes.

The optimization problem is formulated as a 0/1 integer linear program. The following set of 0/1 integer variables ($\forall j \in [1, U], \forall i \in [1, G]$) is defined:

$$I_j(v_i) = \begin{cases} 1 & \text{if variable } v_i \text{ is allocated on mem. unit } j \\ 0 & \text{otherwise} \end{cases}$$

The objective function to be minimized is the total access time of all the memory accesses in the application. For architectures allowing at most one memory access per cycle, the total time is:

$$\sum_{j=1}^{U} \sum_{i=1}^{G} I_j(v_i)[T_{rj} N_r(v_i) + T_{wj} N_w(v_i)] \qquad (1)$$

It is easy to see how the above is the total time for all memory accesses. The term $T_{rj} N_r(v_i)$ is the time for all the read accesses to variable v_i, if it were allocated to memory unit j. A similar term is added for all the write accesses. When multiplied by the 0/1 variable $I_j(v_i)$ the result contributes the memory access time if v_i were indeed allocated to unit j; zero otherwise. Summing this term over all variables (the inner sigma) yields the total access time for a given memory unit. The outer sigma yields the total across all memory units.

For machines that allow at most one memory access per cycle, the formula in (1) is accurate. Most low-end embedded processors we target allow only one memory access per cycle, including some Very Long Instruction Word (VLIW) architectures[3], and hence the formula in (1) is accurate for most targeted chips. For higher-end VLIWs that allow more than one memory access per cycle, however, (1) does not take into account the overlap of memory latencies. To do so requires the formula to include the maximum of latencies of different memory accesses in the same cycle; unfortunately, the objective function does not remain linear since the maximum function is not linear. Thus heuristics instead of optimal 0/1 integer linear solvers must be used; these are not evaluated in this work.

As in any linear optimization problem, a set of constraints is also defined. The first is an exclusion constraint that

[2]The T_{rj},T_{wj} values used for DRAMs are averages. Some modern DRAMs have slightly lower latencies for sequential accesses compared to non-sequential accesses, by about 20%. The difference is not modeled.

[3]VLIWs are architectures that allow the compiler to schedule multiple instructions per cycle, though usually not all of them may be memory instructions.

enforces that for every application variable v_i, it is allocated on only one memory unit:

$$\sum_{j=1}^{U} I_j(v_i) = 1 \qquad (\forall i \in [1, G]) \qquad (2)$$

Another constraint is that the sum of the sizes of all variables allocated to a particular memory unit must not exceed the size of that memory unit:

$$\sum_{i=1}^{G} I_j(v_i) * S(v_i) \leq M_j \qquad (\forall j \in [1, U]) \qquad (3)$$

The objective function (1) combined with the constraints (2) and (3) define the optimization problem. The function and constraints are then solved with an available mathematical solver; we use Matlab [8]. The resulting values of $I_j(v_i)$ are the optimal static allocation.

6. EXTENSION TO STACK VARIABLES

For good performance stack variables – procedure parameters, local variables, and return variables – must be distributed among the different heterogeneous memory units to achieve a more custom allocation. Stack distribution, however, is a complex objective since the stack is normally a sequentially allocated abstraction. Normally, the stack grows in units of stack frames, one per procedure, where a stack frame is a block of contiguous memory locations containing all the variables and parameters of the procedure. The stack grows and shrinks sequentially in units of frames for every nested procedure call with a procedure. Consequent to this sequential abstraction, the entire stack is placed in one memory unit in all programmer-annotated strategies and automatic allocation methods existing today of which we are aware. Custom strategies in which frequently used stack variables are allocated to fast memory, and others to slow memory, have not been investigated.

This paper presents a strategy for distributed stacks applied to heterogeneous memory allocation. Distributed stacks were proposed for the first time in an earlier work by one of the authors [2] for a different purpose for *homogeneous* memory units. For the first time, this paper adapts distributed stacks for heterogeneous memory units. Our approach best explained through an example. On its left, figure 3 shows an example code fragment containing a procedure *foo()* with two local variables a and b. On the right is shown how the stack is distributed into two memory units: variable a is allocated to SRAM, and b to DRAM. This results in two stack pointers, SP_1 and SP_2, in SRAM and DRAM respectively. Both stack pointers must be incremented upon procedure entry and decremented upon exit, instead of one. For ease of use, the implementation of distributed stacks is done automatically by the compiler; the sequential view of an undivided stack is retained for the programmer.

Distributed stacks as in figure 3 incur some software overhead: multiple stack pointers must be updated upon procedure entry and exit, instead of one. Two solutions to overcome the overhead are presented below in alternatives 1 and 2. Alternative 1 is to eliminate the overhead by forcing all the variables within one procedure to be allocated to the same memory level – this way only one stack pointer needs

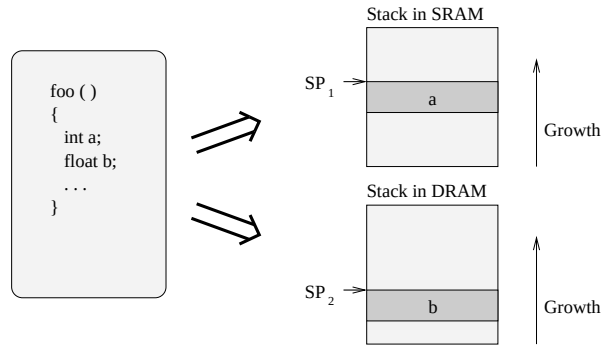

Figure 3: Example of stack split into two separate memory units. Variables a and b are placed on SRAM and DRAM respectively. A call to *foo()* requires the stack pointers in both memories to be incremented.

to be updated per procedure. The stack is still divided, however, as different stack frames can still be allocated to different memory units, and multiple stack pointers exist. Eliminating the overhead has a price, however: grouping together variables results in loss of allocation flexibility. A second solution, presented in alternative 2 below, is to tolerate the overhead, and distribute each individual variable within a stack frame to potentially different banks, as in figure 3. The best solution is to use a hybrid of alternatives 1 and 2: selectively tolerate the overhead for long-running procedures. For long-running procedures, identified by profiling, the impact of a few overhead instructions will be insignificant, so for such procedures the overhead is tolerated. For short-running procedures, each stack frame is allocated to one memory unit, and the overhead is eliminated.

Comparison with globals To see how to modify the global variable formulation for stack variables, consider that the fundamental difference between the two is the limited lifetimes of stack variables. Although stack variables for non-recursive procedures can be treated just like global variables by allocating them for all time, the resulting allocation would not be optimal for stack variables. The reason is that performance with stack variables can be further improved by taking advantage of their limited lifetimes. Stack variables are allocated upon procedure entry and freed upon exit. Thus constraint (3) is no longer valid: the total size of variables allocated to a bank may exceed its size provided not all of them are live at the same time. Variables with non-overlapping lifetimes may share the same space in memory.

One way to incorporate stack variables into our formulation is to treat each stack variable just like a global variable, with the modification that the maximum size constraint (3) is relaxed somewhat. Instead of requiring that all variables fit simultaneously in memory, the call graph of the program is analyzed to construct a new set of constraints that require that only variables that can be live simultaneously fit in memory. The intuition is that one new constraint is introduced for each unique path in the call graph from *main()* to each leaf node in the call graph[4]. The details follow di-

[4]Cycles in the call graph (recursion) cannot be handled this way. Instead they are collapsed to a single aggregate variable in the formulation before this step, and assigned a maximum allowable size based on recursive depth.

rectly from the intuition and are presented below for the two alternative methods 1 and 2, both with their merits.

6.1 Alternative 1: distribution granularity = stack frames

In this first alternative, the stack for each procedure is combined into a single aggregate variable in the formulation. This ensures that the full stack frame for a procedure is allocated to a single memory unit, leading to simplicity in formulation and implementation, and no software overhead for updating multiple stack pointers. The stack is still distributed since different frames could be allocated to different memory units. To describe this formulation, the following symbols are introduced in addition to the ones before for globals:

F = number of aggregate stack variables in the application program(number of functions);
$f_i = i^{th}$ function, $i \in [1, F]$;
$NP(f_i)$ = Total number of unique paths to the function f_i in the call graph;
$P_j(f_i)$ = The jth unique path in the call graph to f_i, $j \in [1, NP(f_i)]$;
$\mathcal{L}$ = The set of all leaf nodes in the call graph.

In addition, $N_r(f_i), N_w(f_i), S(f_i), I_j(f_i)$ are defined as the number of reads to, number of writes to, size, and 0/1 variable for the stack frame for f_i, in an analogous manner to $N_r(v_i), N_w(v_i), S(v_i), I_j(v_i)$. The solution for the $I_j(f_i)$ values yields the desired allocation.

Similar to the formulation for global variables only, the objective function is the total time for all memory accesses (in this case global and stack variables). The objective function for the stack extended formulation is:

$$\sum_{j=1}^{U}\sum_{i=1}^{G} I_j(v_i)[T_{rj}N_r(v_i) + T_{wj}N_w(v_i)] \; + \sum_{j=1}^{U}\sum_{i=1}^{F} I_j(f_i)[T_{rj}N_r(f_i) + T_{wj}N_w(f_i)] \qquad (4)$$

The first term in the above is the original objective function (1) for the global variables, which represents the total time needed to access the global variables. The second term is the total time needed to access the stack variables.

Regarding constraints, the exclusion constraint for global variables presented earlier in (2) is still needed unchanged. A similar constraint is added for stack variables:

$$\sum_{j=1}^{U} I_j(f_i) = 1 \qquad (\forall i \in [1, F]) \qquad (5)$$

As previously mentioned, changes are needed to account for the fact that stack variables can have disjoint lifetimes. Substantial changes are made to the memory size constraint (3) to accommodate the limited lifetimes of stack variables. The new memory size constraint is:

$$\forall j \in [1, U]\ , \forall f_l \in \mathcal{L},\ \forall t \in [1, NP(f_l)], :$$
$$\sum_{i=1}^{G} I_j(v_i)S(v_i) + \sum_{\forall f_p \in P_t(f_l)} I_j(f_p)S(f_p) \leq M_j \qquad (6)$$

The first line of the above states that the second line (the constraint) is replicated for all combinations of memory banks (j), leaf nodes (f_l) in the call graph, and paths to that leaf node (t). The constraint in the second line states that the global variables plus all the stack variables in the given path to the given leaf node must fit into memory. The first term represents the global variable size; the second term represents the size of the stack variables for every call graph path to a leaf function. The stack is of a maximal size when a call-graph leaf is reached; consequently, ensuring that all paths to leaf nodes fit in memory ensures that the program allocation will fit in memory at all times.

The set of constraints in (6) is large as it is replicated across all j, f_l and t. Fortunately, however, this is not expected to adversely impact the runtime of the 0/1 solver by much as the runtime for such solvers depends more on the number of 0/1 variables and less on the number of constraints. Indeed more constraints may decrease the runtime by decreasing the space of feasible solutions.

6.2 Alternative 2: distribution granularity = stack variables

In this alternative, stack variables from the same procedure are allowed to be allocated to different memory units. The formulation is modified as follows. The stack variables are treated just like like global variables in the formulation, leading to an objective function similar to (1). The exclusion constraint is similar to (2). The memory size constraint is similar to (6) with the second $\sum$ function converted to a $\sum\sum$, the outer summation remaining the same as the second summation in (6, and the inner summation summing across all the individual variables in the procedure f_p.

7. EXTENSION TO HEAP DATA

Heap data, allocated in programs by memory routines such as *malloc()*, cannot be allocated optimally by any static method as the frequencies of allocation and size of blocks is unknown at compile-time. Our method is hence a heuristic. The method first allocates memory for malloc calls in SRAM, and then after a certain number of calls, in DRAM. The threshold is profile-determined to try to make SRAM allocation the common case.

The common-case method for heaps is implemented as follows. Each static heap allocation site is treated as an aggregate variable v in the formulation, and profiling is used to count the number of references to data allocated at that site. The variable size is bounded to some desired multiple of the total size of memory allocated at that site, as found by profiling For example, if a *malloc()* site that allocates 20 bytes each time is called 8 times in the profile run, the total size is 160 bytes can be multiplied by a safety factor of 2 to yield a size of 320 bytes. As we shall see, this size is not a hard upper bound but is used to optimize for the common case. The reference counts and sizes are then fed to the linear optimizer as usual and a solution obtained.

To enforce the allocation of malloc sites to their chosen memory levels, memory allocation routines like *malloc()* are cloned, one version for each memory level. Each site then makes a call to the version of *malloc()* for its level. Memory allocation routines for each level maintain separate free memory lists internally. To enforce the size constraint, the maximum allowable size is passed as a parameter for each call site; if it is exceeded at runtime, a memory block from

Benchmark	Source	Total Data Size	Runtime (cycles)	Description
FIR	Trimaran	1036	318K	Finite impulse response algorithm
BMM	Trimaran	120080	5.17M	Block matrix multiplication
FIB	Trimaran	20	1.69M	Computes a Fibonacci number
BTOA	Rutter et. al	411	70K	Changes 8 bit byte streams into printable ASCII

Table 1: Information about the benchmark programs.

a slower and larger memory is returned. This strategy thus optimizes for the common case: when the bound is not exceeded, allocation proceeds in fast memory; otherwise a slow memory is used.

The difficulty with this common-case strategy is that it makes real-time guarantees difficult, as the actual memory unit accessed for a heap access cannot be predicted at compile-time. If such guarantees are needed, then for the purposes of real-time estimation, all heap accesses must be assumed to go to the slowest memory unit to which heap can go to, nullifying much of the advantage of the common-case strategy. Nevertheless the strategy can still be used.

8. RESULTS

Our formulation has been implemented in the public-domain GCC cross-compiler set to target the Motorola M-Core [11] embedded processor. A collection of small kernels and programs, FIR, FIB, BMM, and BTOA have been compiled and evaluated. Their characteristics are shown in Table 1. The benchmarks FIR, FIB, and BMM were taken from the trimaran [4] benchmark suit and BTOA was obtained from [15]. These benchmarks represent code that would be used in typical applications. The first benchmark, FIR, is an implementation of the finite impulse response filter algorithm, which is an important kernel for digital signal processing. Next, the BMM benchmark creates and multiplies two matrices and sums up all the elements of the resulting matrices. The third benchmark, FIB, is a small program that computes a Fibonacci number using a linear recurrence. Finally, the BTOA benchmark is a stream filter to change 8 bit bytes into printable ASCII characters.

Runtimes of the benchmarks are obtained using the M-Core simulator available from Motorola. The M-Core chip simulated has three levels of memory: a 256 Kbyte EEPROM with 1-cycle read and 500-cycle write latency; a 256 Kbyte external DRAM with 10-cycle read/write latency, and an internal SRAM with 1-cycle read/write latency. In the experiments below we analyze the effect of varying the SRAM size while providing ample DRAM and EEPROM memory. The code in these benchmarks was modified by hand to use only global variables. The extension to stack variables is looked at in section 8.1. Small benchmarks were selected for evaluation to make the hand conversion to all global variables easier. The benchmarks did not use the heap. Results for heap data may be investigated in future work. The runtime of the 0/1 optimizer was never a problem: it never took more than a few seconds to run.

The first experiment performed is to compare the runtimes of the benchmarks for three significant cases. First, the runtimes for all the benchmarks are simulated for the case when all the program data is allocated to internal SRAM – the baseline case. Next, the runtime for the case when the amount of SRAM is 20% of the total program data is obtained. Finally, the runtimes of the benchmarks when all the program data is allocated to external DRAM is simulated.

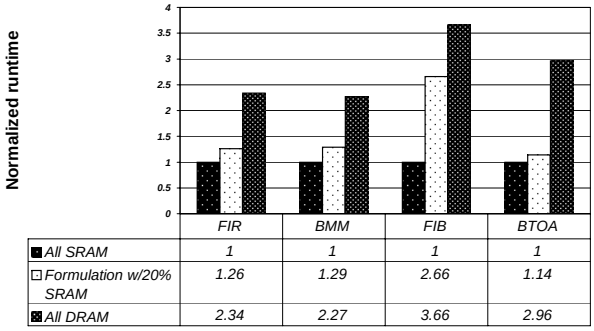

Figure 4: Normalized simulated runtimes for all benchmarks with varying memory configurations. Each group of bars displays the cases, for all benchmarks, when the SRAM size is equal to the program data size, 20% of the program data size (using the formulation), and the case when all program data is placed in DRAM.

The results of this experiment can be seen in Figure 4.

Figure 4 shows that for all the benchmarks except FIB, the formulation, keeping just 20% of the data in SRAM, is able to deliver performance that is closer to that of the all-SRAM case than the all-DRAM case. This demonstrates the overall success of the method in reducing the SRAM size required for good performance to well below the all-SRAM case. To achieve this effect, the formulation uses profile data to place frequently accessed data words in fast memory, and other data to slower memory. In most benchmarks, a large share of the memory accesses go to a small fraction of the data [6]. These accesses can then be placed in fast memory. The FIB kernel is a rare exception – it is a small kernel where only scalars are used, and all the variables have roughly the same number of accesses. Therefore the runtime is adversely impacted when most data goes to DRAM in the formulation. In all cases including FIB, however, our formulation is optimal among all possible static methods for globals and stacks – poor results stem from program characteristics, not any deficiencies in the formulation.

The second experiment performed is to plot the runtimes of the benchmarks when the SRAM size is varied while the EEPROM and DRAM sizes are fixed to an ample size. The results of this experiment can be seen in Figure 5. All the runtimes in Figure 5 are normalized to the 100% SRAM case. The SRAM size is varied from 0% to 100% of the total data size of the program. The data sizes of the FIR, BMM, FIB, and BTOA benchmarks are 1036, 120080, 20, and 411 bytes respectively. The FIR benchmark is composed of a roughly equal number of scalars and arrays; in the BMM benchmark much of the memory is used by a few large arrays needed to store the matrix data; the FIB benchmark is composed entirely of scalar variables; the BTOA benchmark is mostly scalars with one array.

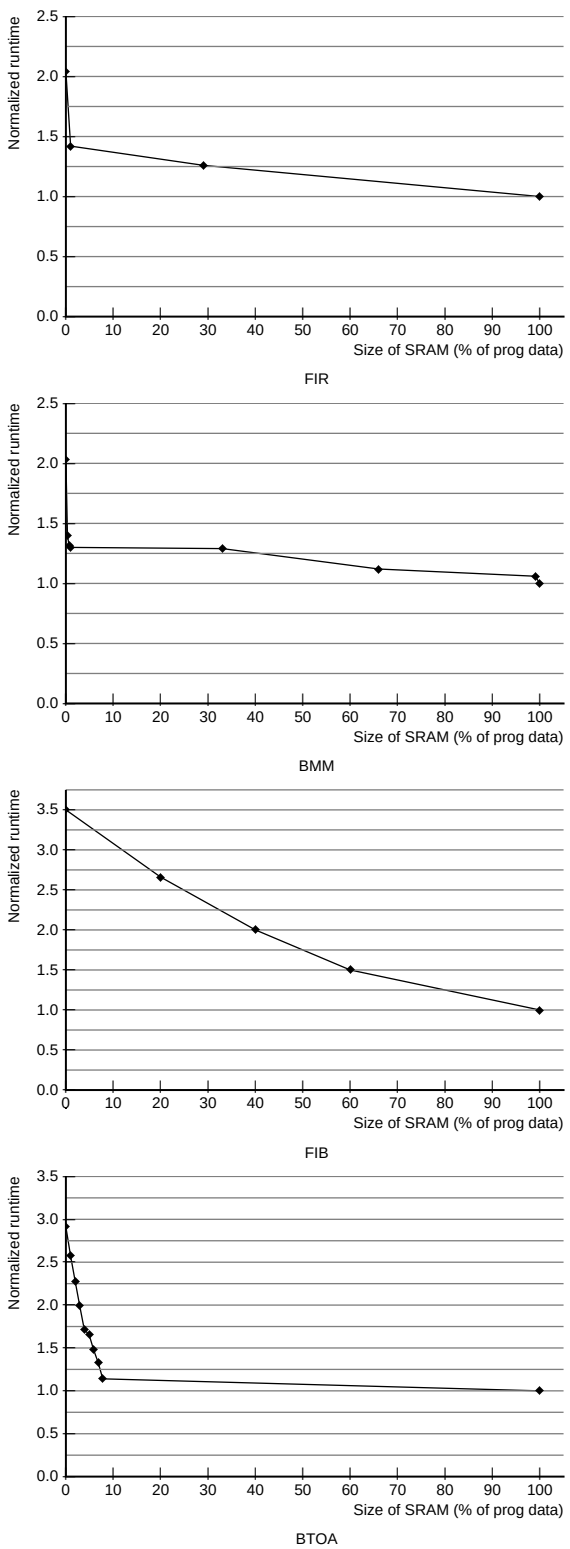

Figure 5: Normalized runtimes for benchmarks with varying SRAM size. DRAM and EEPROM sizes are fixed. X-axis = SRAM size, as a percentage of the total data size for that program. Y-axis = runtime, normalized to 1.0 for SRAM size = 100% of data size. In all cases except FIB note the steep jump in runtime as the SRAM size approaches zero.

In the plots of Figure 5 we see that the runtimes increase as the size of the SRAM decreases, as one would expect. The effectiveness of the formulation can clearly be seen by the large jumps in runtime that occur as the SRAM size gets close to 0%. These jumps happen when the most-often-used variables, that are kept in SRAM as much as possible, are forced into slower memory banks. The formulation guarantees that the runtimes are statically optimal for each memory size. Figure 5 also demonstrates three trends – the BMM numbers are used as an example. *First*, utilizing SRAM can lead to a large gain compared to using only DRAM (runtime 1 vs. 2.27), showing the importance of carefully allocating data to SRAM. *Second*, the formulation is able to get a fairly good runtime (1.30) for even a very small memory size of 32 bytes, compared to the 0 byte case (2.03), showing that the profile-guided optimization is successfully able to put heavily accessed data in SRAM. In the 32 byte case, just 0.03% of the total data is allocated to SRAM, yet it is able to achieve a 36% improvement (2.03 vs. 1.30) in runtime! *Third*, the formulation is successfully able to utilize EEPROM even for data that is written, when SRAM is not available. This is seen for the SRAM=0 case - the runtime reduces to 2.03 (DRAM + EEPROM) compared to 2.27 (only DRAM). This 9% benefit is because the high cost of EEPROM writes may be recovered by frequent reads for data with infrequent writes.

Several benchmark-specific characteristics can also be seen in Figure 5. In the BMM curve the first jumps in runtime occur in large SRAM intervals because this program has many large arrays. The points in the BMM plot indicate where each of the matrix arrays get pushed out of SRAM – and allocated to slower DRAM. The final runtime jumps then occur at very small SRAM sizes – almost 0% SRAM – as the heavily used scalar variables are pushed out of SRAM. This effect can also be seen to a lesser degree in the FIR and BTOA plots. In the FIB plot, since all variables have an equal size, the impact on runtime of moving each variable out of SRAM can more clearly be seen. As the SRAM size is reduced, the lesser-used variables are first allocated to slower memory banks, while the often used variables are kept in memory as long as possible.

8.1 Comparison of alternatives 1 and 2

To see the impact of using Alternative 1 vs. Alternative 2 we measure the runtime of the BMM benchmark, using both alternatives, when the SRAM size is varied while the EEPROM and DRAM sizes are fixed to an ample size. The data for Alternative 1 was generated using the formulation described in Section 6.1. The data for Alternative 2 was simulated using the approach described in section 8 in which stack variables are changed to global variables. A side effect of using this approach is that the overhead incurred by the additional stack manipulation is not accounted for.

Figure 6 summarizes the results of the above experiment. The BMM benchmark was chosen because it contains the most functions of the four benchmarks along with several global variables. From the figure we see that Alternative 1 performed slightly worse than Alternative 2, which was expected. However, the figure indicates Alternative 1 may be a feasible strategy. For the larger SRAM sizes the runtimes are virtually the same while for smaller sizes the performance separation becomes more apparent. The average performance difference between the two alternatives is 11.6%.

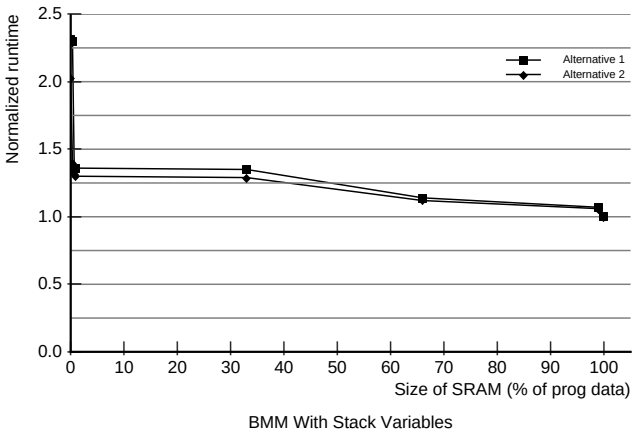

Figure 6: Normalized runtimes of Alternative 1 and Alternative 2 stack formulations for the BMM benchmark with varying SRAM size.

In theory, Alternative 2 will always perform better than Alternative 1 because it provides a finer granularity in the data objects. The finer granularity creates more flexibility for the allocation algorithm, which increases its effectiveness.

9. REAL-WORLD ISSUES

9.1 Application to synthesis

Our compiler method can be used as a synthesis tool to determine the minimum SRAM size needed by the chip during its assembly by a particular user. Using a smaller SRAM can result in significant cost savings. To obtain the smallest SRAM size that still meets the user's performance requirements, the application domain is re-compiled for ever-larger SRAM sizes, and the simulated runtime versus SRAM size is plotted, as in figure 5. The smallest size that delivers the performance needed is chosen.

9.2 Adaption to pre-emptive (context switching) environments

The formulation for automatic data allocation is easily extended to pre-emptive systems that context-switch between multiple programs in a given workload. In context-switching, data from all the programs must be simultaneously present somewhere in memory. If only one program uses all of SRAM, however, the performance of the other programs will suffer. The only way to get better performance is to partition the SRAM among the different programs[5]. In context-switching environments, our framework can partition among different programs in an elegant manner by combining their variables and solving together, after weighing (multiplying) the frequencies $N_r(v_i)$ and $N_w(v_i)$ of each variable by the relative frequency of that context. The solution is optimal for the group of programs when run together.

[5]The alternative solution is to allocate the SRAM to one program only at a time, and save the entire SRAM contents to DRAM on a context switch just like register files. This is infeasible as SRAMs are usually much larger than register files, making the context switch overhead unacceptable.

10. CONCLUSION

This paper presents a compiler method to distribute program data among the different memory units of embedded processors without caching hardware. Without caching, the task of allocating data to the different banks falls to the software. The compiler method derives a static partition, *i.e.*, , one in which the allocation of data to memory units is fixed and unchanging throughout program execution. Among static methods, the method presented is optimal for global and stack data, and a good heuristic for heap data. For the first time, our method distributes stacks among various heterogeneous memory units, resulting in a more custom allocation. The optimality guarantee ensures that the method presented will be as good or better than any programmer derived annotations or competing static compiler technique. The method models the problem as a 0/1 integer linear programming problem and solves it using commercial packages available.

We are encouraged by the preliminary results obtained. They show the benefits of optimal allocation: with just 20% of the data in SRAM, our method is able to decrease the runtime by 39% on average for our benchmarks vs. allocating all data to slow memory, without any programmer involvement. For some programs, less than 5% of data in SRAM achieves a similar speedup. The variance in results is due to application characteristics. Applications that access some data more frequently than other data see large decreases in runtime when those data are allocated to fast memory – the amount of runtime reduction depends on the relative frequencies of access.

11. REFERENCES

[1] A. Appel and L. George. Optimal Spilling for CISC Machines with Few Registers. In *Proceedings of the SIGPLAN '01 Conference on Program Language Design and Implementation*, Snowbird, UT, June 2001.

[2] R. Barua, W. Lee, S. Amarasinghe, and A. Agarwal. Compiler Support for Scalable and Efficient Memory Systems. *IEEE Transactions on Computers, Special Issue on Advances in High Performance Memory Systems*, November 2001.

[3] S. S. Bhattacharyya, R. Leupers, and P. Marwedel. Software Synthesis and Code Generation for Signal Processing Systems. *IEEE Transactions on Circuits and Systems*, 47(9), September 2000.

[4] T. T. Consortium. The Trimaran benchmark suite. *Available at http://www.trimaran.org/*, 1999.

[5] G. Hallnor and S. K. Reinhardt. A fully associative software-managed cache design. In *Proc. of the 27th Int'l Symp. on Computer Architecture (ISCA)*, Vancouver, British Columbia, Canada, June 2000.

[6] J. Hennessy and D. Patterson. *Computer Architecture A Quantitative Approach.* Morgan Kaufmann, Palo Alto, CA, second edition, 1996.

[7] T.-C. Lee, V. Tiwari, S. Malik, and M.Fujita. Power Analysis and Minimization Techniques for Embedded DSP Software. *IEEE Transactions on VLSI Systems*, Mar. 1997.

[8] *Matlab 6.1.* The Math Works, Inc., 2001. http://www.mathworks.com/products/matlab/.

[9] C. A. Moritz, M. Frank, and S. Amarasinghe.

FlexCache: A Framework for Flexible Compiler Generated Data Caching. In *The 2nd Workshop on Intelligent Memory Systems*, Boston, MA, November 12 2000.

[10] *CPU12 Reference Manual*. Motorola Corporation, 2000. http://e-www.motorola.com/brdata/PDFDB/-MICROCONTROLLERS/16_BIT/-68HC12_FAMILY/REF_MAT/CPU12RM.pdf.

[11] *M-CORE - MMC2001 Reference Manual*. Motorola Corporation, 1998. http://www.motorola.com/SPS/-MCORE/info_documentation.htm.

[12] New York City, Office of Budget and Management. *Website on frequently asked questions on linear programming*. http://www.eden.rutgers.edu/~pil/FAQ.html, New York, NY, 1999.

[13] P. R. Panda, N. D. 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), July 2000.

[14] P. Paulin, C. Liem, M. Cornero, F. Nacabal, and G. Goossens. Embedded Software in Real-Time Signal Processing Systems: Application and Architecture Trends. *Invited paper, Proceedings of the IEEE*, 85(3), March 1997.

[15] P. Rutter, J. Orost, and D. Gloistein. BTOA: Binary to printable ASCII converter source code. *Available at http://www.bookcase.com/library/software/-msdos.devel.lang.c.html*.

[16] J. Sjodin, B. Froderberg, and T. Lindgren. Allocation of Global Data Objects in On-Chip RAM. *Compiler and Architecture Support for Embedded Computing Systems*, December 1998.

[17] *TMS370Cx7x 8-bit microcontroller*. Texas Instruments, Revised Feb. 1997. http://www-s.ti.com/sc/psheets/spns034c/spns034c.pdf.