Storage Allocation for Embedded Processors*

Jan Sjödin
Uppsala University
Department of Information Technology
Box 337
SE-751 05 Uppsala
Sweden
jans@csd.uu.se

Carl von Platen
IAR Systems
Malmö Slagthus
Carlsgatan 12 A
SE-211 20 Malmö
Sweden
carl.von.platen@iar.se

ABSTRACT

In an embedded system, it is common to have several memory areas with different properties, such as access time and size. An access to a specific memory area is usually restricted to certain native pointer types. Different pointer types vary in size and cost. For example, it is typically cheaper to use an 8-bit pointer than a 16-bit pointer. The problem is to allocate data and select pointer types in the most effective way. Frequently accessed variables should be allocated in fast memory, and frequently used pointers and pointer expressions should be assigned cheap pointer types. Common practice is to perform this task manually.

We present a model for storage allocation that is capable of describing architectures with irregular memory organization and with several native pointer types. This model is used in an integer linear programming (ILP) formulation of the problem. An ILP solver is applied to get an optimal solution under the model. We describe allocation of global variables and local variables with static storage duration.

A whole program optimizing C compiler prototype was used to implement the allocator. Experiments were performed on the Atmel AVR 8-bit microcontroller [2] using small to medium sized C programs. The results varied with the benchmarks, with up to 8% improvement in execution speed and 10% reduction in code size.

1. INTRODUCTION

Most embedded systems have an irregular memory organization. By this, we mean that the memory consists of several memory areas with different characteristics. For example, *on-chip memory* (small and fast internal memory) is a common feature where frequently used data can be stored to improve program performance. Similarly, it is common to have different native pointer types to access different types of memories. By effectively using these types of memories and smaller (cheaper) pointer types, both execution speed and program size can be improved. This, in turn, affects other important parameters such as energy consumption, production cost and even the physical size of the system.

The allocation problem consists of:

- Allocating each static variable of a program in a memory segment.
- Assigning a native pointer type to each pointer expression of the same program.

Native pointer type assignment is dependent on the storage allocation. We must use pointer types that can access the memory areas where data has been stored. If allocation is done first, without taking the pointer types into account, the code quality may not be as good as if both are done simultaneously.

Currently, programmers must manually locate variables and select native pointer types. This is typically done by annotating code using non-standard mechanisms, such as pragmas or keywords. It is a time-consuming task and it renders the source code non-portable. If an application is moved to a different platform, a new allocation must be done and the source code must be modified.

We propose a model that is capable of describing architectures with irregular memory organization and with several native pointer types. From this model we can derive an integer linear program (ILP). A solution to the ILP directly corresponds to a solution to the allocation problem. Under the assumptions made in the model, the solution is optimal.

In order to perform variable allocation, all memory accesses in the program must be known. If only parts of the user code is provided it is not possible to modify the addressing mode used to access the variables. If a restricted pointer type is used, the set of objects a pointer may point to, known as the *points-to set*, must be allocated in the memory accessible by that pointer type. For example, if we have disjoint address spaces, the entire points-to set must be allocated in the same space. If an object in the points-to set is located outside the memory accessible by a pointer, the resulting program is incorrect. Thus, pointers are able to impose restrictions on how we can allocate variables. This means that whole program optimization (WPO) is required both for the allocation and the native pointer type assignment.

*This work has been conducted under the Whole Program Optimization for Embedded Systems project. This is an ongoing project at Uppsala University, in collaboration with IAR Systems, and is sponsored by the Advanced Software Technology (ASTEC) competence center.

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

Storage allocation in on-chip memory has previously been explored by Panda *et al.* [10], as a solution to reduce cache conflicts. The algorithm used was heuristic and did not take code size into consideration. Similarly, Cooper and Harvey [4] evaluate the use of on-chip memory as a spill area for register allocation to minimize the risk of cache conflicts. Memory bank and register allocation for ASIPs using simulated annealing was done by Sudarsanam and Malik [14]. There has been a lot of work done on storage allocation (using graph coloring) for DSPs [9, 11], where the data layout is critical for pre- and post-increment/decrement memory operations. ILP has been used by Goodwin and Wilen [5], and Appel and George [1] for spilling strategies for register allocation. Kandemir *et al.* [8] used ILP for optimizing cache locality. Bixby *et al.* [3] describe a 0-1 integer programming solution to data layout of arrays for parallel programs.

The authors have previously described allocation in on-chip memory as a 0-1 knapsack problem [12]. This is a special case of the model described in this paper.

2. MEMORY ORGANIZATION

A feature found in most embedded processors is on-chip memory, which can be accessed efficiently. In contrast, access to external memory requires external buses to be driven and, typically, causes latency. Another common feature is a specific address range, known as the *zero page*. Data located in the zero page can be accessed using fast and compact code since smaller pointer types can be used. The "near" segment of a segmented memory architecture is similar in that access within this segment is fast and compact; whereas, access to "far" data generally requires specific code for bank switching. Another common example is modified Harvard architectures. In this case it is beneficial to allocate constant data in the program address space. Different native pointer representations and addressing modes are used for data in the program and data address spaces.

Some architectures have *separate address spaces*. Separate address spaces result in disjoint sets of pointer types, where the pointers in one set cannot access the memory areas accessible in any other set. All data that may be accessed by a pointer p must be located in the address space that p can access. To allocate data for these types of processors we must know the points-to sets to ensure that no pointer restrictions are violated. The hardware may use separate address buses to access separate physical memories. This is usually reflected in the instruction set by several types of load and store instructions that also takes different pointer types as operands.

In contrast, if we have a *single address space*, there is always a general native pointer type that may access the entire memory. There is no restriction on where the variables are allocated since general pointers may always be used. However, we may want to use more restricted pointer types to optimize the program. In this case, information about the points-to set is needed to determine if a more restricted pointer type can be used.

Consider the Atmel AVR, an 8-bit microcontroller. The AVR typically has between 0 and 4KB on-chip memory. Figure 1 shows the memory configuration of an AVR with 4KB internal memory. It has a single address space. The shaded area represents the internal memory. An access to internal memory takes 2 clock cycles, while an access to external memory with zero wait states takes 3 clock cycles. We have

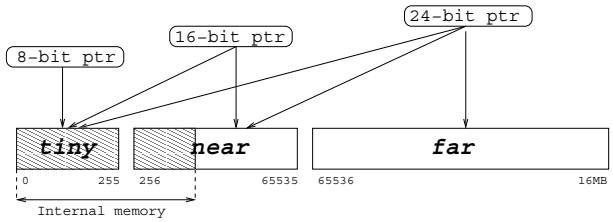

Figure 1: AVR RAM Configuration

```
( 0)??main_0:
( 1)   MOV     R16,R24 ;
( 2)   MOV     R17,R25 ;
( 3)   LSL     R16     ;
( 4)   ROL     R17     ;
( 5)   MOV     R30,R16 ;
( 6)   MOV     R31,R17 ;
( 7)   SUBI    R30,LOW((-(b) & 0xFFFFFF)) ; [b]
( 8)   SBCI    R31,HIGH((-(b) & 0xFFFFFF)); [b]
( 9)   LDI     R19,(b) >> 16   ; [b]
(10)   OUT     0x3B,R19        ;
(11)   LD      R20,Z   ; b
(12)   LDD     R21,Z+1 ; (b + 1)
(13)   MOV     R16,R24 ;
(14)   MOV     R17,R25 ;
(15)   LSL     R16     ;
(16)   ROL     R17     ;
(17)   MOV     R30,R16 ;
(18)   MOV     R31,R17 ;
(19)   SUBI    R30,LOW((-(a) & 0xFFFFFF)) ; [a]
(20)   SBCI    R31,HIGH((-(a) & 0xFFFFFF)); [a]
(21)   LDI     R19,(a) >> 16   ; [a]
(22)   OUT     0x3B,R19        ;
(23)   ST      Z,R20   ; a
(24)   STD     Z+1,R21 ; (a + 1)
(25)   ADIW    R25 : R24,1     ;
(26)   CPI     R24,10  ;
(27)   LDI     R16,0   ;
(28)   CPC     R25,R16 ;
(29)   BRLT    ??main_0    ; [??main_0] BRANCH
```

Figure 2: Array copy with arrays in the *far* segment.

three native pointer types: 8-bit, 16-bit and 24-bit. Using the 8-bit pointer may require only one instruction compared to three instructions with the 24-bit pointer. Figure 2 shows an assembly listing of a simple copy from one array `b` to another array `a`, where both arrays have been allocated in the *far* segment. The loop contains 29 instructions. Figure 3 shows the same loop, but the arrays are located in the *tiny* segment and is only 19 instructions. By using the *tiny* segment, code size is decreased, since cheaper addressing modes can be used. Speed is improved since fewer instructions are executed and we save one clock cycle per memory operation since on-chip memory is accessed.

3. AN ABSTRACT MEMORY MODEL

The memory model is described by a number of *memory segments*, which represent *subsets* of the total memory space. There is also a number of pointer types, and a relation between the pointer types and the memory segments. Each memory segment is uniform with respect to properties, such as the memory speed and the set of admissible addressing modes.

```
(0)??main_0:
(1)   MOV     R16,R24 ;
(2)   LSL     R16     ;
(3)   MOV     R30,R16 ;
(4)   SUBI    R30,(-(b) & 0xFF)   ; [b]
(5)   LDI     R31,0   ;
(6)   OUT     0x3B,R31        ;
(7)   LD      R20,Z   ; b
(8)   LDD     R21,Z+1 ; (b + 1)
(9)   MOV     R16,R24 ;
(10)   LSL     R16     ;
(11)   MOV     R30,R16 ;
(12)   SUBI    R30,(-(a) & 0xFF)  ; [a]
(13)   ST      Z,R20   ; a
(14)   STD     Z+1,R21 ; (a + 1)
(15)   ADIW    R25 : R24,1     ;
(16)   CPI     R24,10  ;
(17)   LDI     R16,0   ;
(18)   CPC     R25,R31 ;
(19)   BRLT    ??main_0    ; [??main_0] BRANCH
```

Figure 3: Array copy with arrays in the *tiny* segment.

Non-uniform address spaces are modeled by a subdivision into memory segments which are uniform with respect to all properties relevant to storage allocation. The memory model of a target architecture is defined in the following way:

Let $M_1, M_2, ..., M_S$, be the set of memory segments of a target architecture. The size of a memory segment M_j, denoted $Size(M_j)$, is the number of addressable units of storage (typically bytes) in M_j. Let $P_1, P_2, ..., P_T$ be the set of pointer types, for the same target architecture. The set of memory segments to which a pointer type P_k can refer to is given by

$$Mem(P_k) = \{j \mid P_k \textit{ can point to } M_j\}$$

Any number of additional properties can be associated with the memory segments and the pointer types. The exact form of these properties, which are used to formulate the feasibility and the cost of a given storage allocation, is implementation dependent.

3.1 Example: AVR

In this example, we model the memory of an AVR with 4KB on-chip memory as described in Section 2. The three pointer types are 8-bit (P_1), 16-bit (P_2) and 24-bit (P_3). The AVR also has native pointer types for ROM or flash memory, but to simplify the presentation, we will not consider these. Data memory is divided into 4 segments: one segment M_1 (internal) that may be accessed by an 8-bit pointer (M_1), two segments M_2 (internal) and M_3 (external) for the 16-bit pointer accesses and M_4 for 24-bit accesses. Table 1 shows the memory model.

3.2 Example: 8051

Intel 8051 [6] has three separate address spaces: internal data memory, external data memory and code memory. There are three native pointer types: an 8-bit pointer to the internal memory (P_1), a 16-bit pointer to the external memory (P_2) and a 16-bit pointer to the code memory (P_3). Figure 4 shows the memory organization and the native pointer types. For convenience, a fourth pointer type (P_4), that is capable of accessing any of the three address

M_j	$Size(M_j)$	P_k	$Mem(P_k)$
$M_1(tiny)$	256	P_1	$\{M_1\}$
$M_2(near)$	3480	P_2	$\{M_1, M_2, M_3\}$
$M_3(near)$	61440	P_3	$\{M_1, M_2, M_3, M_4\}$
$M_4(far)$	64K-16M		

Table 1: Atmel AVR Memory Model with 4K internal RAM

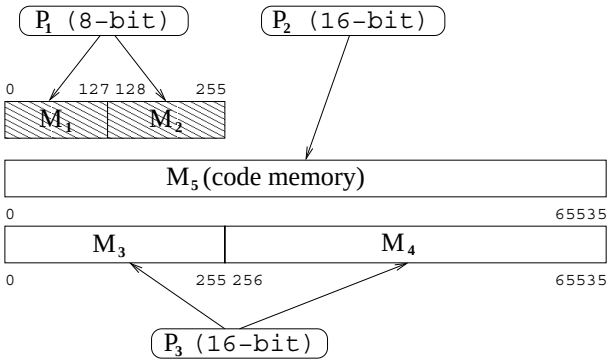

Figure 4: 8051 Memory Configuration

spaces, is emulated in software using a 24-bit pointer format. Using a 24-bit pointer type is extremely expensive. The internal data memory space is not uniform and must be split into two memory segments. The first 128 bytes (M_1) can be accessed very efficiently using the direct addressing mode, whereas the remaining 128 bytes (M_2) can only be accessed using register indirect addressing. Similarly, there is a 256-byte segment (M_3) of the external memory space that is accessed more efficiently than the rest of the 64K byte address space (M_4). Code memory (M_5) can only be used for constant data. The 8051 benefits from storage allocation in the same way as the AVR. Cheaper pointers and on-chip memory can be used to improve program performance.

3.3 Example: ARM

A target system based on an ARM [7] core may have both fast, on-chip memory and significantly slower external memory. The memory organization would be modeled by two segments, on-chip memory (M_1) and external memory (M_2). A single pointer type (P_1) is capable of accessing the entire address space.

Storage allocation has little or no impact on the code size of the ARM-based system. Apart from possible differences in instruction scheduling, the exact same code sequence is generated regardless of storage allocation. In this case, the

M_j	$Size(M_j)$	P_k	$Mem(P_k)$
M_1	128	P_1	$\{M_1, M_2\}$
M_2	128	P_2	$\{M_3, M_4\}$
M_3	256	P_3	$\{M_5\}$
M_4	65280	P_4	$\{M_1, M_2, M_3, M_4, M_5\}$
M_5	65536		

Table 2: Intel 8051 Memory Model

objective instead is to allocate frequently accessed variables in the fast on-chip memory.

4. COST MODEL

The program is modeled by the set of variables appearing in the program and the statements of the intermediate representation which are relevant for storage allocation. We assume that the source program is represented in a form similar to three-address code and that the instructions are typed. Thereby, the pointer operations can be separated from integer operations. We consider only the instructions that are relevant for the variable allocation problem along with the variables of the program.

Let $V = \{v_1, ..., v_N\}$, be the set of variables referred to in the source program. The size of a variable v, denoted $Size(v)$, is the number of allocation units (bytes) required by v.

Let $STMTR$, be the set of individual occurrences of three-address instructions which are relevant for the problem of variable allocation. These would be the memory operations (loads and stores) and instructions in where the assigned register is a pointer (pointer arithmetic and address calculations).

Let $E = \{e_1, ..., e_M\}$, be the set of pointer expressions. The function *PtrExp* maps the instructions in $STMTR$ to a pointer expression in E. We assume that *PtrExp* is defined in a way that allows the client of the storage allocator to modify the type of a single pointer expression independently.

A solution to the storage allocation problem consists of two parts: the memory segment of each variable, $v \in V$, denoted $Seg(v)$ and a pointer type for each pointer expression, $e \in E$ denoted $PtrT(e)$.

We assume that the cost of a particular solution can be expressed as the sum of the cost contribution of each variable and each pointer expression treated in isolation. We will demonstrate that this model is insufficient in some cases. A generalization of the model, which handles these cases, is discussed in Section 5.1.

For each pointer expression $e \in E$, each pointer type $t \in P$, and each statement $S \in STMTR$ we let $ptrcost_S(e, t)$ denote the cost contribution of selecting type t for e in statement S. Similarly, we denote the cost contribution for each variable v located in memory segment m with $varcost_S(v, m)$, where $S \in STMTR$.

To allow accurate modeling of addressing modes and other machine idioms for pointer operations, we assume that a target-dependent matching is performed on the intermediate code. In this way, several intermediate statements may be combined into a single operation that subsumes the individual statements and contributes to the overall cost only once. A particularly common machine idiom is the absolute (or direct) addressing mode by which a variable is accessed directly. This also reduces the total number of variables needed in the binary integer program described in Section 5.

The cost of a particular solution of the storage allocation problem is defined in terms of the cost contribution of each statement S in $STMTR$.

$$Cost(S) = \sum_{e \in E} ptrcost_S(e, PtrT(e)) + \sum_{v \in V} varcost_S(v, Mem(v))$$

When modeling a static cost, such as the size of a program, the sum of all statements in $STMTR$ is formed. When modeling a dynamic cost, such as the execution time, the cost contribution first needs to be scaled by the (estimated) execution frequency of the statement.

5. BIP FORMULATION

The storage allocation problem is formulated as a binary integer program (BIP). This formulation is based on the model of a processor's memory organization and the cost model.

A feasible solution of the storage allocation problem has to satisfy two conditions. First, the total size of the variables placed in a single memory segment must not exceed the size of the segment. Second, any pointer expression that may point to a particular variable must be of a type that can access the memory segment where the variable has been allocated. Among the feasible solutions, we are interested in one with minimal cost. It is possible to express both static costs (code size) and, using execution profiles, dynamic costs (execution time).

A binary integer program (BIP) follows directly from the definition of feasible solutions and the cost of a particular solution. Each variable v_i of the source program is represented by S boolean variables $x_{i1}, ..., x_{iS}$ in the BIP, where S is the number of memory segments. The meaning of $x_{im} = 1$ is that variable v_i is allocated in memory segment M_m. We require each variable to be placed in exactly one memory segment, that is:

$$x_{i1} + ... + x_{iS} = 1, \quad 1 \leq i \leq N$$

where N is the number of source-program variables. Furthermore, we require the total size of the variables in a particular segment to be no greater than the size of the segment.

$$Size(v_1)x_{1m} + ... + Size(v_N)x_{Nm} \leq Size(M_m),$$
$$1 \leq m \leq S$$

Each pointer expression e_j is represented by T boolean variables $y_{j1}, ..., x_{jT}$, where T is the number of pointer types. The intuition is that of $y_{jk} = 1$ when pointer expression e_j has type P_k. We require each pointer expression to have a unique type, that is:

$$y_{j1} + ... + y_{jT} = 1, \quad 1 \leq j \leq M$$

where M is the number of pointer expressions. We also require the type of each pointer expression e_j to be sufficiently general for all the variables in its points-to set:

$$Seg(v) \in Mem(PtrT(e_j)), v \in Pt(e_j)$$

or

$$\sum_{i \in Pt(e_j)} \sum_{m \in Mem(P_k)} x_{im} \geq |Pt(e_j)|\, y_{jk}$$
$$1 \leq j \leq M, \quad 1 \leq k \leq T$$

The summation is performed over all variables that are in the points-to set of the pointer expression e_j in all memory segments, which a P_k -pointer can access. Since each variable is located in exactly one memory segment, the sum can, at the most, be the size of the points-to set; $y_{jk} = 1$ does not violate the constraint in this case. If some variable in the points-to set is not accessible by a P_k -pointer, y_{jk} is zero.

Pointer Type	Segment M_1	Segment M_2
P_1	2 bytes	3 bytes
P_2	4 bytes	6 bytes

Table 3: Correlated Costs

The objective function (to be minimized) is the cost of the solution, which can be expressed as a linear combination of the variables x_{im} and y_{jp}:

$$\sum_{i=1}^{N}\sum_{m=1}^{S} a_{im}x_{im} + \sum_{j=1}^{M}\sum_{k=1}^{T} b_{jk}y_{jk}$$

where

$$a_{im} = \sum_{S \in STMTR} varcost_S(v_i, M_m),$$

and

$$b_{jk} = \sum_{S \in STMTR} ptrcost_S(r_j, P_k),$$

5.1 Discussion

The scalability of the proposed optimization is of major concern. The problem size depends on the number of variables and the number of pointer expressions. Both of which are likely to grow linearly in the size of the source program. The number of memory segments and the number of pointer types are, however, parameters of the target architecture and can be considered constant. Although the BIP-formulation presented in this paper can be improved, we expect that finding an optimal solution for large programs is impractical in general. Finding a heuristic allocation algorithm is important if the allocator should be implemented in a production compiler.

The accuracy of the cost model is essential to the quality of the solution. Fundamental to the model is that the cost of a single statement can be expressed as a linear combination of cost contributions from variables and pointer expressions. There is a particularly common case that cannot be modeled precisely in this way. Some target architectures may represent pointers using multiple machine words. Loading a pointer from memory may thus require several instructions. The cost of this operation depends on the size (type) of the pointer to be loaded and the memory segment of the variable from which the pointer is loaded. Consider the pointer types shown in Table 3: P_1 takes one load instruction, and P_2 takes two load instructions. The size of the instructions are 2 and 3 bytes for the memory segments M_1 and M_2, respectively. We see that the cost cannot be expressed as a linear combination. The cost model can be extended to reflect cost contributions of pairs of pointer expressions and variables. In BIP-formulation, the pairs would be represented by auxiliary variables. In the example given above, one auxiliary variable per pointer-typed load operation would be required.

We also note that, by modifying the model slightly, we can represent address spaces with different units of addressable storage. A common case for Harvard architectures is that the data space is byte addressable, and the program space is addressable in units of instruction words. In this case, the *Size* function would be dependent not only on the variable, v_i, but also on the memory segment M_m.

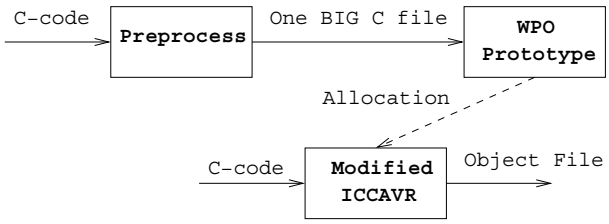

Figure 5: Compiler Framework

6. IMPLEMENTATION

In this section, we describe how we implemented the allocation. Figure 5 shows the compiler framework. We have implemented a whole program optimization framework to do the analysis and to solve the allocation problem. The information is then read by a modified production compiler (ICCAVR) for the AVR microcontroller in order to generate executable code. This approach eliminated transformations that can can arise when other optimizations get a larger scope, e.g., inlining. This would have occurred if we had generated code from the WPO framework directly.

6.1 WPO Prototype

To analyze an entire application, we have implemented a whole program optimization compiler (ICCWPO) based on an IAR Systems production compiler. We added a preprocessing stage that resolves all preprocessor definitions (e.g., macros) and concatenates the resulting files into one large file containing all code. ICCWPO reads the file and produces an allocation, which is then used by a modified IAR Systems AVR compiler.

6.1.1 Points-to Analysis

We needed a good estimate of the points-to set of each pointer in the program. If we have an unknown function, it may modify all global pointers and all pointers passed to the function. Since whole program analysis was used, this problem was eliminated. The points-to analysis [13] used is an interprocedural unification-based, flow-insensitive analysis. There is no information about which fields of structured data may be accessed through a pointer; each complex object is treated as a unit. Calls to library procedures with known effects are treated as special cases.

6.1.2 ILP Solver

We have integrated *lp_solve* into our prototype compiler. *lp_solve* is a freely available LP/MILP solver written by Michel Berkelaar at Eindhoven University of Technology. Information about all global variables and pointers in the program is collected and put into *lp_solve* to get an allocation.

6.2 Modified ICCAVR

The modified ICCAVR compiler compiles the original C code and uses the allocation information produced by the WPO compiler to modify the memory attributes of the allocated variables. The allocation information is used during the parsing stage to modify the memory attributes of the

variables. The result is essentially the same as if the source code was annotated with keywords.

7. EXPERIMENTS AND RESULTS

To evaluate the performance of our allocation scheme, we measured the execution time and code size of 6 benchmarks. The code was compiled with the prototype compiler using both speed and size optimization. The allocation does not include constant strings since these are rarely used. The results were compared with code compiled on the standard ICCAVR compiler. The code was executed in a simulator to get the execution times (in clock cycles).

The AVR microcontroller comes in several different versions, where the size of the internal memory can vary from 0 to 4KB or more. We wanted to see the effects of having different sizes of internal RAM. As reference points, we also measured "internal" memory sizes of 64KB (entire *near* memory segment) and 16MB (full address space). With 64KB internal memory, we may include part of the program stack, or even the entire stack, depending on the total size of the global variables. With 16MB internal memory, all accesses are counted as internal.

Table 4 shows a list of our benchmarks. Appendix A has information of where to obtain the benchmarks and a short description of what they do. The tables in Appendix B show the results. The results vary a lot with the benchmarks. This is expected since some programs are CPU intensive, while others are memory intensive.

7.1 Code Size Results

The code size decreased slightly for most benchmarks. The one notable exception is the Statemate program. When the program was optimized for size, the code size was 90% compared to non-allocated code. When optimized for speed, the result was 73%. The Statemate program is quite special since it was automatically generated. It has very many direct accesses to global variables which all can be allocated at lower memory addresses. This allows us to use fewer instructions for address calculations to access the global variables.

A few benchmarks had slightly increased code size for allocated code. The reason for the size increase is that the code factoring optimization finds fewer identical instruction sequences; therefore, the code cannot be compressed as much.

7.2 Execution Time Results

The execution times were improved for most benchmarks. The best result came from the Statemate benchmark shown in Table 6. When the program was optimized for speed, there was an 8% speedup compared to the non-allocated code. For size optimized code, the result was 20%. The reason for the high speedup on size-optimized code is partly because of more memory accesses since less code is inlined and common sub-expression elimination cannot eliminate redundant loads. We can, however, see a correlation between execution times for speed and size-optimized programs. Both Table 6 and Table 5 show that good speedup on speed-optimized code give better speedup on size-optimized code. This can be very useful since the execution time penalty for optimizing size can be reduced with a good allocator.

Name	Lines	# Globals	Total Size (bytes)
`MM`	93	4	15002
`Minver`	206	7	220
`Jfdctint`	382	1	126
`Anagram`	655	10	267
`KS`	835	14	22564
`Statemate`	1274	106	212

Table 4: Benchmarks

8. FUTURE WORK

In the future, we would like to look at a number of issues regarding our storage allocation framework.

- We would like to verify how exact our model is in predicting the improvements in execution speed and code size reduction.
- An ILP solver is generally not practical. Therefore, we would like to explore different heuristic methods for storage allocation.
- We would like to implement the allocator on a microcontroller with separate address spaces. If no allocation is done, the compiler may only use the default address space. Therefore, we could possibly gain more on these types of architectures.
- The points-to analysis we used could possibly prevent some allocation opportunities. A more precise analysis could improve this.
- Many programs do not have large amounts of global data. We would like to extend the allocation to include for example local variables and possibly individual fields in structs.

9. CONCLUSION

In this paper, we have described a model of memory hierarchies for storage allocation which is applicable to a wide range of embedded processors. We have used this model to implement a memory allocator using ILP to get an optimal allocation with respect to the model.

By taking advantage of the different memory segments and native pointer types, we can improve both program speed and size. The experimental results show a speedup of up to 8% for speed-optimized programs. Code size was not affected very much except for programs with many static references to global data. The best result was a benchmark with a 10% smaller code size.

The tedious task of manually allocating data has been eliminated. By automatically allocating data we improve code quality without sacrificing portability. Code robustness is also improved because the programmer no longer needs to include target-specific information in the source code.

10. REFERENCES

[1] A. Appel and L. George. Optimal spilling for CISC machines with few registers. *ACM SIGPLAN Notices*, 36(5):243–253, May 2001.

[2] Atmel Corporation. *The AVR Instruction Set.* `http://www.atmel.com/atmel/acrobat/doc0865.pdf`.

[3] R. Bixby, K. Kennedy, and U. Kremer. Automatic data layout using 0-1 integer programming. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, volume A-50, pages 111–122, 1994.

[4] K. D. Cooper and T. J. Harvey. Compiler-controlled memory. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 100–104, San Jose, California, 1998.

[5] D. W. Goodwin and K. D. Wilken. Optimal and near-optimal global register allocation using 0-1 integer programming. *Software—Practice and Experience*, 26(8):929–965, Aug. 1996.

[6] Intel Corporation. *Embedded Microcontrollers and Processors Vol. I*, 1992.

[7] D. Jaggerl. *ARM Architecture Reference Manual*. Addison-Wesley, 2000.

[8] M. Kandemir, P. Banerjee, A. Choudhary, J. Ramanujam, and E. Ayguade. An integer linear programming approach for optimizing cache locality. In *Proceedings of the 1999 Conference on Supercomputing*, pages 500–509, 1999.

[9] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, and A. Wang. Storage assignment to decrease code size. *ACM Transactions on Programming Languages and Systems*, 18(3):235–253, May 1996.

[10] P. R. Panda, N. D. 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, Jan. 1997.

[11] A. Rao and S. Pande. Storage assignment optimizations to generate compact and efficient code on embedded DSPs. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 128–138, Atlanta, Georgia, 1999.

[12] J. Sjödin, B. Fröderberg, and T. Lindgren". Allocation of global data object in on-chip ram. Presented at the CASES'98 workshop, `http://www.capsl.udel.edu/conferences/cases99/cases98/paper04.ps`.

[13] B. Steensgard. Points-to analysis in almost linear time. In *Proceedings 23rd SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1996.

[14] A. Sudarsanam and S. Malik. Simultaneous reference allocation in code generation for dual data memory bank ASIPs. *ACM Transactions on Design Automation of Electronic Systems.*, 5(2):242–264, Jan. 2000.

APPENDIX

A. Benchmarks

The following list describes each benchmark.

`MM` - Matrix multiplication

`Minver` - Matrix inversion

`Jfdctint` - JPEG integer implementation of the forward discrete cosine function

`Anagram` - Finds anagrams in a dictionary

`KS` - Kernighan-Schweikert graph partitioning

`Statemate` - Automatically generated code by the STAtechard Real-time-Code generator STARC.

`Anagram` and `KS` were taken from the *Pointer-Intensive Benchmark Suite*, by Todd Austin at:
`http://www.cs.wisc.edu/~austin/austin.html`.

The other benchmarks were obtained from the benchmark repository for WCET analysis at C-LAB:
`http://www.c-lab.de/home/en/download.html#wcet`.

B. Tables

The tables show the execution times and code sizes for six benchmarks. Rows labeled S show the results for program compiled for speed with the standard compiler. SA shows the results for a program compiled for speed and variable allocation with the modified compiler. Z and AZ is for size-optimized compilation. Rows labeled *Speedup* shows the speedup for allocated programs and rows labeled *Code Size* show the code size ratio allocated code compared to regular code.

MINVER Runtimes (clock cycles)								
	Internal Memory Size (bytes)							
	0	**256**	**512**	**1024**	**2048**	**4096**	**64K**	**16M**
S	55879	55879	55879	55879	55879	55771	52879	52879
AS	55044	54059	54059	53915	53915	53915	52435	52435
Speedup	*1.0152*	*1.0337*	*1.0337*	*1.0364*	*1.0364*	*1.0344*	*1.0085*	*1.0085*
Z	71566	71566	71566	71566	71458	71458	68676	68676
AZ	63179	64037	64037	63893	63893	63893	62497	62497
Speedup	*1.1327*	*1.1176*	*1.1176*	*1.1201*	*1.1184*	*1.1184*	*1.0989*	*1.0989*
MINVER Code Size (bytes)								
S	5537							
AS	5131	5127	5127	5127	5127	5127	5127	5127
Code Size	*0.9267*	*0.9260*	*0.9260*	*0.9260*	*0.9260*	*0.9260*	*0.9260*	*0.9260*
Z	4595							
AZ	4503	4527	4527	4527	4527	4527	4527	4527
Code Size	*0.9800*	*0.9852*	*0.9852*	*0.9852*	*0.9852*	*0.9852*	*0.9852*	*0.9852*

Table 5: Minver

STATEMATE Runtime (clock cycles)								
	Internal Memory Size (bytes)							
	0	**256**	**512**	**1024**	**2048**	**4096**	**64K**	**16M**
S	24867	24867	24867	24867	24867	24867	21852	21852
AS	23890	23157	23157	22969	22969	22969	20698	20698
Speedup	*1.0409*	*1.0738*	*1.0738*	*1.0826*	*1.0826*	*1.0826*	*1.0558*	*1.0558*
Z	29176	29176	29176	29176	29176	29176	26131	26131
AZ	25935	24392	24392	24204	24204	24204	21095	21095
Speedup	*1.1250*	*1.1961*	*1.1961*	*1.2054*	*1.2054*	*1.2054*	*1.2387*	*1.2387*
STATEMATE Code Size (bytes)								
S	9481							
AS	7551	6957	6957	6957	6957	6957	6957	6957
Code Size	*0.7964*	*0.7338*	*0.7338*	*0.7338*	*0.7338*	*0.7338*	*0.7338*	*0.7338*
Z	6577							
AZ	6443	5979	5979	5979	5979	5979	5979	5979
Code Size	*0.9796*	*0.9091*	*0.9091*	*0.9091*	*0.9091*	*0.9091*	*0.9091*	*0.9091*

Table 6: Statemate

JFDCTINT Runtime (clock cycles)								
	Internal Memory Size (bytes)							
	0	**256**	**512**	**1024**	**2048**	**4096**	**64K**	**16M**
S	162202	162202	162202	162202	162202	162202	159240	159239
AS	160325	159305	159305	159305	159305	159305	157623	157622
Speedup	*1.0117*	*1.0182*	*1.0182*	*1.0182*	*1.0182*	*1.0182*	*1.0103*	*1.0103*
Z	175830	175830	175830	175830	175830	175830	172794	172793
AZ	173672	172652	172652	172652	172652	172652	170816	170815
Speedup	*1.0124*	*1.0184*	*1.0184*	*1.0184*	*1.0184*	*1.0184*	*1.0116*	*1.0116*
JFDCTINT Code Size (bytes)								
S	4571							
AS	4441	4441	4441	4441	4441	4441	4441	4441
Code Size	*0.9716*	*0.9716*	*0.9716*	*0.9716*	*0.9716*	*0.9716*	*0.9716*	*0.9716*
Z	3519							
AZ	3463	3463	3463	3463	3463	3463	3463	3463
Code Size	*0.9841*	*0.9841*	*0.9841*	*0.9841*	*0.9841*	*0.9841*	*0.9841*	*0.9841*

Table 7: Jfdctint

ANAGRAM Runtimes (clock cycles)								
	Internal Memory Size (bytes)							
	0	**256**	**512**	**1024**	**2048**	**4096**	**64K**	**16M**
S	3008.7K	3008.7K	3008.7K	3008.7K	3008.7K	3008.7K	2745.9K	2622.6K
AS	2997.7K	2980.7K	2980.7K	2980.4K	2980.4K	2980.4K	2716.8K	2607.1K
Speedup	*1.0037*	*1.0094*	*1.0094*	*1.0089*	*1.0095*	*1.0088*	*1.0107*	*1.0059*
Z	4111.8K	4111.8K	4111.8K	4111.9K	4111.8K	4111.7K	3837.0K	3713.7K
AZ	4055.0K	3974.4K	3974.4K	3974.2K	3974.2K	3974.1K	3699.8K	3590.1K
Speedup	*1.0140*	*1.0346*	*1.0346*	*1.0346*	*1.0346*	*1.0346*	*1.0371*	*1.0344*
ANAGRAM Code Size (bytes)								
S	8054							
AS	7948	7956	7956	7956	7956	7956	7956	7956
Code Size	*0.9868*	*0.9878*	*0.9878*	*0.9878*	*0.9878*	*0.9878*	*0.9878*	*0.9878*
Z	6898							
AZ	6938	6952	6952	6952	6952	6952	6952	6952
Code Size	*1.0058*	*1.0078*	*1.0078*	*1.0078*	*1.0078*	*1.0078*	*1.0078*	*1.0078*

Table 8: Anagram

KS Runtimes (clock cycles)								
	Internal Memory Size (bytes)							
	0	**256**	**512**	**1024**	**2048**	**4096**	**64K**	**16M**
S	136.75M	136.75M	136.75M	136.75M	136.75M	136.75M	119.24M	116.62M
AS	136.74M	136.72M	136.72M	136.70M	136.70M	136.68M	118.68M	116.32M
Speedup	*1.0001*	*1.0002*	*1.0002*	*1.0004*	*1.0004*	*1.0005*	*1.0047*	*1.0025*
Z	238.89M	238.89M	238.89M	238.89M	238.89M	238.89M	221.27M	218.64M
AZ	239.57M	239.55M	239.55M	245.30M	245.29M	242.02M	224.28M	221.92M
Speedup	*0.9971333*	*0.9972186*	*0.9972*	*0.9739*	*0.9739*	*0.9870*	*0.9866*	*0.9852*
KS Code Size (bytes)								
S	15448							
AS	15320	15320	15320	15322	15322	15344	15310	
Code Size	*0.9917*	*0.9917*	*0.9917*	*0.9918*	*0.9918*	*0.9933*	*0.9911*	*0.9911*
Z	12304							
AZ	12240	12240	12240	12276	12276	12332	12314	12314
Code Size	*0.9948*	*0.9948*	*0.9948*	*0.9977*	*0.9977*	*1.0023*	*1.0008*	*1.0008*

Table 9: KS

MM Runtime (clock cycles)								
	Internal Memory Size (bytes)							
	0	**256**	**512**	**1024**	**2048**	**4096**	**64K**	**16M**
S	26742K	26742K	26742K	26742K	26742K	26742K	24175K	24175K
AS	26742K	26722K	26722K	26722K	26722K	26722K	23972K	23972K
Speedup	*1.0000*	*1.0007*	*1.0007*	*1.0007*	*1.0007*	*1.0007*	*1.0085*	*1.0085*
Z	33487K	33487K	33487K	33487K	33487K	33487K	31425K	31425K
AZ	33532K	33512K	33512K	33512K	33512K	33512K	31267K	31267K
Speedup	*0.9987*	*0.9992*	*0.9992*	*0.9992*	*0.9992*	*0.9992*	*1.0051*	*1.0051*
MM Code Size (bytes)								
S	2637							
AS	2643	2643	2643	2643	2643	2643	2641	2641
Code Size	*1.0022753*	*1.0022753*	*1.0022753*	*1.0023*	*1.0023*	*1.0023*	*1.0015*	*1.0015*
Z	2479							
AZ	2483	2483	2483	2483	2483	2483	2481	2481
Code Size	*1.0016*	*1.0016*	*1.0016*	*1.0016*	*1.0016*	*1.0016*	*1.0008*	*1.0008*

Table 10: Matrix Multiply