High Performance Code Generation through Lazy Activation Records

M. Satpathy[1], R.N. Mahapatra[2], S. Choudhuri[2] and S.V. Chitnis[3]

[1]Applied Software Engineering Group
Department of Computer Science, University of Reading, Reading RG6 6AY, England
[2]Hardware Software Codesign Group
Department of Computer Science, Texas A&M University, College Station, TX 77843-5534
[3]CMG Wireless Data Solutions B.V.
Merweplein 5, 3432 GNβ Nieuwegein, The Netherlands
m.satpathy@reading.ac.uk, {rabi,choudhri}@cs.tamu.edu, sachin.chitnis@cmg.com

Abstract

For call intensive programs, function calls are major bottlenecks during program execution since they usually force register contents to be spilled into memory. Such register to memory spills are much more pronounced in presence of recursion. A function call is usually accompanied by the creation of its activation record at function entry. In this paper, we will deviate from this usual practice; we create an activation record only when we find it necessary. The result is that on many occasions we can execute a function call without actually creating its activation record. We call our strategy lazy activation record strategy *(LARS) and show how this strategy is particularly important for call–intensive programs. The LARS subsumes many traditional techniques like leaf-call optimization and tail-recursion optimization, and in addition, it extends Chow's shrink-wrapping in terms of scope and granularity.*

We also demonstrate how the LARS can be an effective optimization strategy in case of battery operated embedded systems, since not only it can reduce execution time but also energy consumption.

Keywords. *Register Utilization, Activation Records, Embedded Systems, Low Power Optimization.*

1 Introduction

During program execution, whenever a function is called, the runtime environment usually creates an activation record (sometimes called function frames or activation frames, or simply frames) in the system stack to store all the entities in relation to the function activation; such entities include the return address, function arguments, function result and some locations to store temporary computations. Creation of a frame is a costly process, since it involves a number of memory accesses. Therefore, for call–intensive programs in general, and recursive programs in particular, the problem of reducing memory accesses due to function calls is of paramount importance. In this paper, we will refer to functions and procedures interchangeably.

Recursion is an elegant technique to represent many problems occurring in practice; to name a few, they are tree traversal, sorting, Fast Fourier Transforms etc. Presence of a recursive call prohibits efficient use of machine registers since registers need to be spilled at call boundaries. Optimization techniques like function unfolding or inlining aims at reducing the overhead due to function calls. This is a standard technique which is usually applied over some non-recursive functions. In presence of recursion, inlining will be of little help since it will increase the code size without removing any call from the code. Therefore, inlining over recursive calls are less often attempted [7].

Embedded systems that run on batteries need to consume as less energy as possible. One way of achieving this is to minimize memory references and if possible to use register operations in their place. Usually the compilation strategies for recursive programs do not achieve such an objective because of the associated complexity.

In this paper, we will discuss a compilation strategy which will minimize memory references due to the creation of activation records. The strategy is not to create activation frames at call boundaries but to delay their creation as much as possible; the result is that in many cases, we may avoid creating them. We will call this strategy the *lazy activation record strategy* (LARS). The LARS can be applied over all kinds of programs, be they recursive or non-recursive; how-

ever, it is more effective for recursive programs. We will demonstrate the efficacy of our strategy first through a running example and then through some benchmark programs.

The organization of the paper is as follows. Section 2 discusses the related work. In Section 3, we introduce the basic idea behind the lazy activation records through a few motivating examples. Section 4 discusses the code generation strategy. In Section 5, we present some benchmark results. In section 6, we analyse and summarize the main results of our paper. Section 7 concludes the paper.

2 Related Work

A tail-recursive call is often converted into iteration and thereby the procedure call overhead associated with tail-recursive calls can be avoided. A tail-call is a general case of tail-recursion. A call from procedure f() to procedure g() is a tail call if immediately after g() returns, f() also returns; i.e. there is no other computation between the two returns. In such a case, the tail call can be compiled so that instead of making a call to the callee, a $jump$ can be made to the code segment of the callee, and furthermore, a single $return$ instruction will be sufficient for both the caller and the callee.

A leaf-routine is a procedure which is a leaf in the call graph of a program. Leaf-routine optimization tries to optimize the procedure prologue and epilogue overhead which are usually associated with non-leaf procedures. If it can be statically estimated that a leaf-call can be fully computed using available registers, then the stack frame may not be created [14]. Smaller leaf-routines are often inlined.

Davidson and Holler [7] have done extensive study on subprogram inlining which is the technique of unfolding function calls so that either they are eliminated or are replaced by larger pieces of straight-line code. Function unfolding increases code size. And unfolding is usually avoided in presence of recursion.

Chow's shrink wrapping technique assumes that the available register set is partitioned into caller–save and callee–save register subsets [5]. His technique is usually applied to callee-save registers of a procedure when they need to be used inside the procedure body. Instead of storing callee-save registers at procedure entry and restoring them at procedure exit, they are placed only along relevant control paths inside the procedure body; i.e. control paths which do not require to use them do not have to include the save/restore operations. For optimal placement of the register save/restore operations, Chow uses data-flow analysis technique similar to the one used my Morel and Renvoise [13] in the context of partial redundancy elimination.

Satpathy et al [15] have discussed a method of optimizing register spills in presence of recursion by keeping multiple versions of the same function. Since this approach increases code size, the number of versions are usually kept small. It is to be noted that their strategy is defined in relation to functional programs which exhibit a large number of function calls.

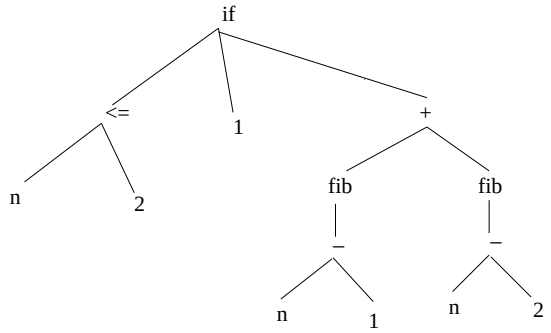

Figure 1. Graph of the Fibonacci Function

Chakrapani *et al.* [4] have developed a precise model of a state-of-the-art embedded processor which they have used in conjunction with a robust compiler to conduct experiments on the impact of various compiler optimizations from the viewpoint of power consumption. Based on this, they have proposed a taxonomy of compiler optimizations which classifies them into three classes. The Class A optimizations benefit energy due to improvement in performance; i.e. due to reduction in the number of machine cycles. Such optimizations include reductions in the number of loads and stores, procedure cloning, loop transformations, procedure inlining, partial redundancy eliminations etc. Class B optimizations are those which benefit energy but they don't have any impact on performance. They include instruction scheduling, register pipelining etc. Class C optimizations can have negative impacts both on performance and energy consumption. The conclusion is that the highest impact on energy consumption occurs from improving the performance of the code. In this paper, our optimization strategy falls into the Class A category.

EMSIM (Energy Simulation Framework for an Embedded Operating System) [17] is a simulation framework for analysing the energy consumption characteristics of an embedded system featuring the embedded Linux Operating System running on the StrongARM processor [2]. This will be our framework for analysing energy consumption and performance of our benchmark programs.

3 The Basic principle

We will illustrate the basic idea behind lazy activation records (LAR) through a few motivating examples.

3.1 Example 1:

Figure 1 shows the graphical representation of the Fibonacci example. When the predicate $n \leq 2$ is $true$, the function returns 1 as the result. Assume that when this invocation was done, the return address and the lone argument were in registers. Then the computation of $n \leq 2$ could be done in registers, and when the predicate of the if–statement is true, we just need to return the result value of 1 in a register. Thus we have managed to compute this invocation of fib without creating its activation record. Of course if the control took the else–branch, we would have no other option but to create the activation record. Therefore, we can delay the creation of the activation record till the program control reaches the else–branch. In other words, whenever control took the then–branch, i.e. for all terminating calls, we could manage without creating activation records. A call to Fibonacci function with argument n results in the creation of $fib(n)$ number of calls, which is exponential in relation to n. Therefore, we have been able to eliminate $\lceil fib(n)/2 \rceil$ number of activation frames.

It is worth mentioning that some customized compilers for embedded systems like the C compiler for the ARM processor [16, 2] do such an optimization in very few simple cases like the Fibonacci function where the terminating branch returns a constant; for example, it does no such optimization if the terminating branch returns a variable as the result. However, our strategy, as we will demonstrate, is much more general and it avoids creation of activation records in much more complex scenarios.

3.2 Example 2:

Ack(x,y) = if ($x == 0$) then return ($y + 1$)
else if ($y == 0$) then
return Ack($x - 1, 1$)
else
return Ack($x - 1$, Ack($x, y - 1$))

This is the Ackermann function and we consider this example because it illustrates many interesting scenarios of the LARS. When a call to $Ack(x, y)$ is made, assume the return address and the arguments are in registers. When control takes the terminating branch of the outer if–statement, following the argument of the previous example, no activation record will be created for the above call to Ack. This is because, the predicate ($x == 0$) and ($y + 1$) both could be evaluated in registers. Further, when control takes the then–branch of the inner if–statement, then notice that it is a tail–recursive call. In such a case, using tail–recursion optimization, we can manage without creating an activation record for the original call to Ack. We will illustrate this scenario in the next section. Activation record for the call to Ack would be created only if the control took the else–branch of the inner–if.

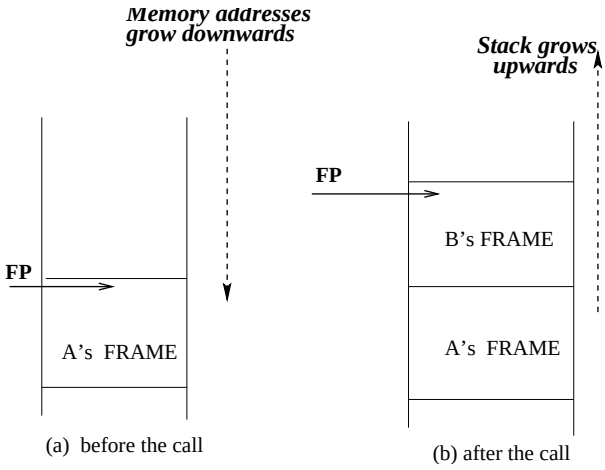

Figure 2. The Stack Organization

4. The Compilation Strategy

4.1 Assumptions

1. There are special registers $R_{a1}, R_{a2}, \ldots, R_{ak}$ for passing arguments. If the number of arguments exceed k, then the additional arguments are passed via stack, and in such a case, an activation record must be created.
2. A special register R_{ret} stores the return address.
3. A special register R_{res} stores the result of a call.
4. The general purpose registers are labelled as $R_1, R_2, \ldots, R_n$
5. An activation record is pointed to by a special register FP (for Frame Pointer). Within it, the return address is stored in the very first location, in the next few locations the arguments are kept, and in subsequent locations the temporary computations and other items are stored. The $Load$ and $Store$ instructions access a frame with FP as the base register. As shown in figure 2, the memory addresses grow downwards, whereas the stack grows upwards. $fsize$, the size of an activation record, is determined statically.

Note that many of the RISC processors conform to the first four assumptions; one such example is the ARM processor which is mostly used for embedded systems [16]. We have taken the last assumption for generating and explaining our code in this paper; otherwise, this assumption could be modified to suit any architecture.

4.2 Special Instructions

Under the above assumptions, we will define the semantics of the following three instructions as follows:

```
Call <fun-label>:
            Move R_ret,PC    //store return address
            Jump <fun-label> //jump to funct. label

Return_1:   Move PC,R_ret    // recover return address

Return_2:   Load R_ret,FP    // return address in R_ret
            Add  FP, fsize   // discard frame
            Move PC,R_ret    // continue
```

What the above means is that the $Call$ instruction just stores the return address in R_{ret} and makes a jump to the function label; the activation record, if necessary, would be created explicitly by separate instructions. There are now two return instructions: $Return_1$ and $Return_2$. $Return_1$ is executed when the activation record for the called function has not been created, and the $Return_2$ is executed when the activation record has been created. The former only retrieves the return address from R_{ret}, and the latter does the same in addition to discarding the activation record.

4.3 Code for Some Example Programs

Let us now generate code for the following program. The meaning of the instructions other than the special instructions defined above should be obvious. In the following, FP[i] means the i-th location in relation to the base address FP.

$fib(n) = if\ (n \leq 2)\ then\ 1$
$else\ fib(n-1) + fib\,(n-2)$
Main: $fib\ 10$

//The Generated Code:

```
Main:   Move R_a1, 10  // Move 10 to argument register.
        Call fib       // Make call to fib
        Stop
fib:    Cmp R_a1,2     //Compare
        Jgt Else       //Jump if greater than
        Move R_res,1
        Return_1
Else:   Sub FP, fsize  // Create frame
        Store FP, R_ret  // Store return address
        Store FP[1], R_a1 //Store argument
        Sub R_a1,1
        Call fib
        Store FP[2], R_res//Store temporary result
        Load R_a1,FP[1]
        Sub R_a1,2
        Call fib
        Load R_1,FP[2]
        Add R_res,R_1    //Result in R_res
        Return_2
```

In the code above, if the control took the $then$–branch, then no activation frame would be created; all computation would be done in registers. On the other hand, if control took the else–branch, then an activation record would be created. Note that we make a conservative decision about creating frames; i.e. we avoid the creation of a frame along a control path if we are sure that it will not be needed; and in absence of such information we create a frame. Creation of a frame may be followed by transfer of the return address, the parameters or other register contents to the frame. The code for the Ackermann Function, which has been shown below, illustrates an interesting scenario in which the the LAR strategy gets combined with the tail recursion optimization.

//Code for Ackermann Function

```
//Before the call, arguments are in R_a1 and R_2.
Ack:    Cmp R_a1, 0
        Jne Else_1       // Jump if not equal
        Move R_res, R_a2
        Add R_res, 1
        Return_1
Else_1: Cmp R_a2,0
        Jne Else_2
        Sub R_a1,1
        Move R_a2,1
        Jump Ack         //tail recursion; hence jump
Else_2  Sub FP, fsize    //Create frame
        Store FP, R_ret  // return address in frame
        Store FP[1], R_a1/argument in frame
        Sub R_a2,1
        Call Ack
        Load R_a1, FP[1]
        Sub R_a1, 1
        Move R_a2, R_res
        Call Ack
        Return_2
```

In the code above, there is a tail recursive call in the $Else_1$ branch and it has been replaced by a $jump$. Furthermore, notice that the second call to Ack in the $Else_2$ is also a tail recursive call; therefore, using the knowledge of the architecture, this call also can be replaced by a $Jump$ instruction.

4.4 Formalization of the Strategy

The LARS is based on the fact that we can manage to compute a function (or a control path within the function) without creating the frame of the function. In this context, the following issues need to be addressed:

1. How to know that a control path can be computed without creating the activation record?

2. If an activation record has to be created where the instruction for creating the record would be placed?

3. There may be many control paths which require references to activation records. Should we place frame creation operations along all the paths?

The formalization of our LARS will be done over the control flow graph (CFG) of the function. Analysis over the control flow graph would answer the questions that we have raised above.

Figure 3 shows the control flow graph of the Ackermann function. We have selected this function because it is simple and it presents most of the interesting scenarios of the LARS. The correspondence between the code of the Ackermann function in Section 4.3 and the present CFG should be obvious. The dashed edges in the CFG signifies that the last instruction of the source basic block (source of the dashed edge) is a function call and control flows to the destination basic block after the function in the source basic block returns. The back edge in the CFG has resulted because of tail-recursion in the then–branch of the inner if–statement (refer to the program in Section 3.2) has been converted to an iterative loop. Note that a CFG has some special basic blocks called $entry$ and $exit$; a CFG has exactly one $entry$ basic block but it may have more than one $exit$ basic blocks. A control path in the graph of a function definition is defined as a path from the $Entry$ basic block to the $Exit$ basic block that the computation can take in relation to the CFG of the function because of control flow. We will now present some definitions.

Minimal Control Path (MCP): In a CFG, remove the back edges. Then starting from the exit node(s), traverse along the opposite direction of the edges towards the entry basic block. The control paths when seen in the reverse order of the above traversals are defined as the *Minimal Control Paths* (MCPs). Figure 3 has two MCPs, shown in dashed lines.

Minimally Independent Control path (MICP): If two MCPs form a diamond between them, i.e. they share two basic blocks and in between the shared blocks there is a non-shared basic block, then both such MCPs belong to the same MICP. However, the descendant basic block in the diamond should not be an $exit$ basic block because, in such a case it could be viewed as two MICPs. In Figure 5 (a), the two MCPs are in the same MICP, whereas in (b) the two MCPs are in different MICPs.

Maximally Independent Control Region (MICR): The region of the maximum number of basic blocks within a MICP such that it does not overlap with any other MICP. For the purpose of finding MICRs, it is assumed that if a MICP includes a basic block of a loop then the whole loop is a part of the MICP. In the figure 3, the only loop is a part of both the MICPs. The two MICRs of the same figure are shown in Figure 4 as the regions in dashed lines.

We will classify the MICPs into the following categories:

- Type 1: The MICP neither has any function call in it, not it is associated with any loop.
- Type 2: The MICP has one or more loops but it does not have any function call (MICP 1 in Figure 3).
- Type 3: The MICP has more than one function calls but it ends with a tail call (like the MICP 2 in in Figure 3). If there was only one call and it was in tail position then it would have already been replaced by a jump, as in case of the *Jump Ack* instruction in the only loop in the same figure.
- Type 4: The MICP has function calls but none in tail position.

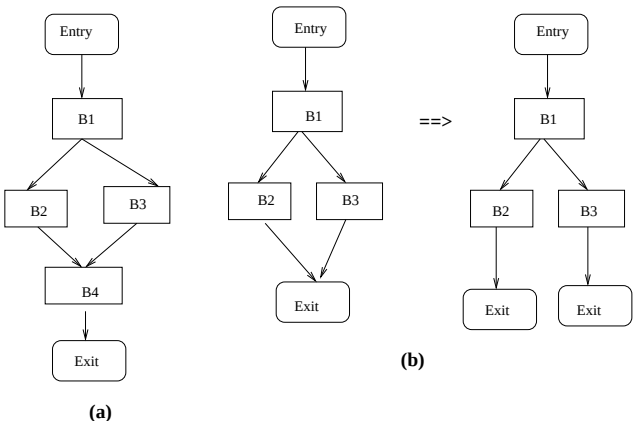

Figure 5. (a) Two MCPs within the same MICP, (b) Two MCPs in different MICPs

With the above definitions, we will formalize our strategy. For each MICP, we will find out if the whole MICP can be computed by registers alone without any reference to the activation record. If so, we will not create a frame for the MICP; otherwise, a frame will be created.

We will assume that our algorithm, which decides whether an activation record would be created in a MICP, is performed after global register allocation. Presently, we will assume that the coloring algorithm by Chow and Hennessy [6] is used for this purpose though any other standard algorithm could be used. By the end of register allocation, code has already been more or less generated. Our algorithm makes one more pass over the CFG to decide whether

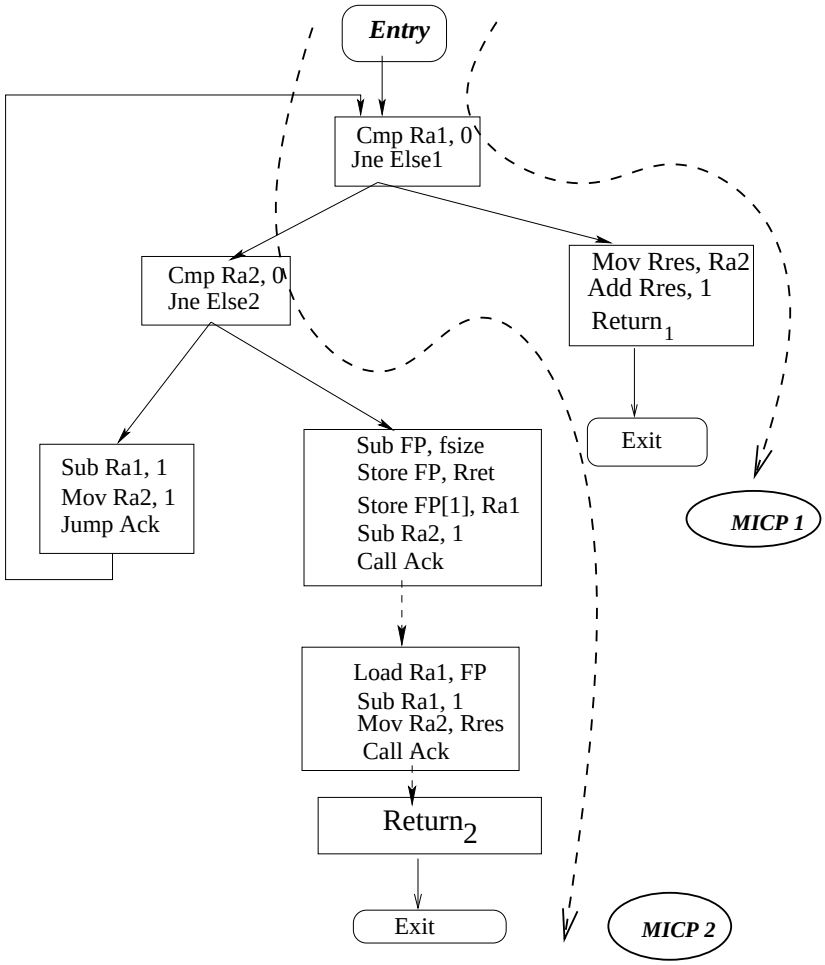

Figure 3. Control Flow Graph of the Ackermann Function

a MICP needs an activation record. Furthermore, our strategy does some engineering at this stage so that a MICP requiring an activation record, in some cases, can be converted into one not requiring such a record, an issue we will discuss later.

One issue that needs to be sorted out is the placement of instructions for creating activation records in the code. We will place such instructions, in relation to each MICP, at the point where control enters into the MICR. Therefore, in the following we will concentrate on the code fragment of the MICR of a MICP.

We will also assume that, for each MICP, the basic blocks in the region (MICP - MICR) can be computed in registers; i.e. there are no memory references. If not, then it is trivial to see that the MICP requires a frame, and we need not make any more analysis on this MICP.

Algorithm: *CodeForLazyActivationRecords.*
Input: *CFG of the procedure F.*
Output: *Code for the MICRs.*

Step 1: Find all MICPs and MICR for each MICP.
Step 2: for each MICP do.
{
Based on MICP type generate code as:
Case MICP of type
Type 1: /* *no frame* */
If MICR requires no memory reference
< code for the MICR>
Return$_1$
Type 2: /* *no frame* */
If MICR requires no memory reference
< code for the MICR>
Return$_1$
Type 3: /**frame necessary* */
< code for creating activation record>
.
< code for the non–tail call>
.
<code after tail call optimization>

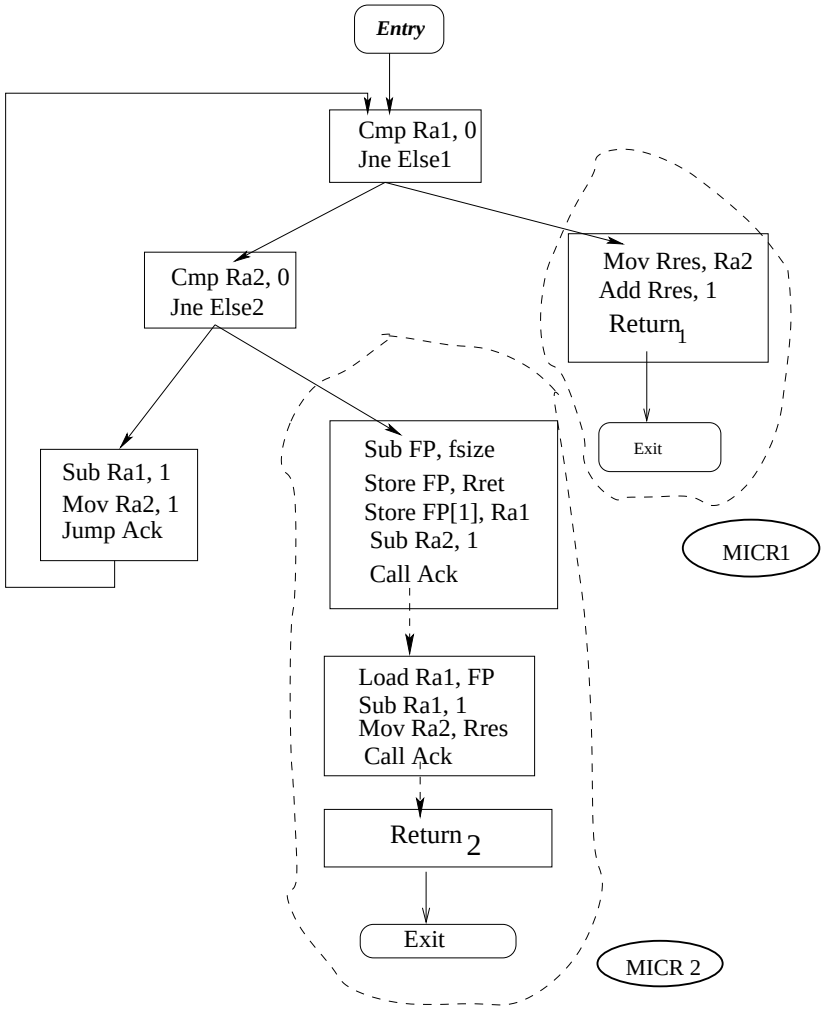

Figure 4. MICRs in the CFG

<code for tail-call optimization>
Jmp <function–label>
Type 4: /* *frame necessary* */
< code for creating activation record>
.
Return$_2$
}

Step 3: *for* each MICR of Types 1 and 2, if it cannot be executed with available registers, generate code as:
< code for creating activation record>
< code for the MICR>
Return$_2$

Step 4: Stop

4.5 Computing a MICP with available registers

We have talked about generating code for a MICP using the available registers. In the present context, by available registers we mean the registers that are available for code generation within the current function. Such registers are dependent on the machine architecture. In general terms, they mostly constitute:

- the argument registers
- the scratch pad registers if they are available
- caller save registers

As we have discussed earlier, our analysis is performed after the global register allocation phase. So, if at this point in time, the basic blocks in the MICP have no memory references and there are no function calls, then our job is done; we will not create the activation for the MICP. However, if

there were still a few memory references, then we could still assign some registers to them as in the following case.

The priority-based graph coloring approach by Chow and Hennessy [6] in the context of RISC architectures reserves a few registers (4 integer and 4 floating point registers) for use in evaluating expressions that involve variables. This is cited as a drawback of the priority-based approach because the strategy without knowing in advance how many registers should be reserved for this purpose, it reserves the maximum number of registers that may be needed [14]. In reality all may have not been used. Therefore such registers, and any other, if available could be allocated to eliminate the remaining memory references in a MICP and then it may not need the activation record.

In short, if we could manage to execute a control path by means of the above register set, then we would not generate activation records for the concerned control path; otherwise, we will.

4.6 Placement of frame creation instructions

Let us call a MICP that does not require an activation record is called a $good$ MICP; otherwise, we will call it a bad MICP. It may so happen that there may be a number of bad MICPs in a CFG; in such a case, if we place frame creation instructions along each individual path then then it may unnecessarily increase the code size. To alleviate this problem, if two MICPs are bad, then we will try to place a single instruction in a common ancestor block such that no good MICP falls within their scope. For finding a common ancestor, we have to traverse along the corresponding MCPs of both the MICPs in reverse direction. However, if we encounter a good MICP in between, then we may have to introduce multiple frame creation instructions, a scenario illustrated in Figure 6.

In the figure, MICP 1 is good whereas MICP 2 and 3 are bad. If the instruction for creating a frame was placed in block B_2, then it would be the common instruction for both the MICPs, 2 and 3. However, MICP 1 is good; therefore, the instruction cannot be placed in B_2. That is is why we require two separate instructions for creating activation records, one for MICP 2 and the other for MICP 3, and they may be placed at their entry points. Since blocks B_5 and B_7 are within loops, we place such instructions by creating new blocks at points shown by shaded circles in the figure.

5 Experimental Results

Table 1 shows some benchmark programs which we have considered to measure the efficacy of LARS. Fibonacci, Ackerman and the Tak are the standard benchmark recursive programs which are highly call-intensive. FFT is the Fast Fourier Transform program which uses the recursive

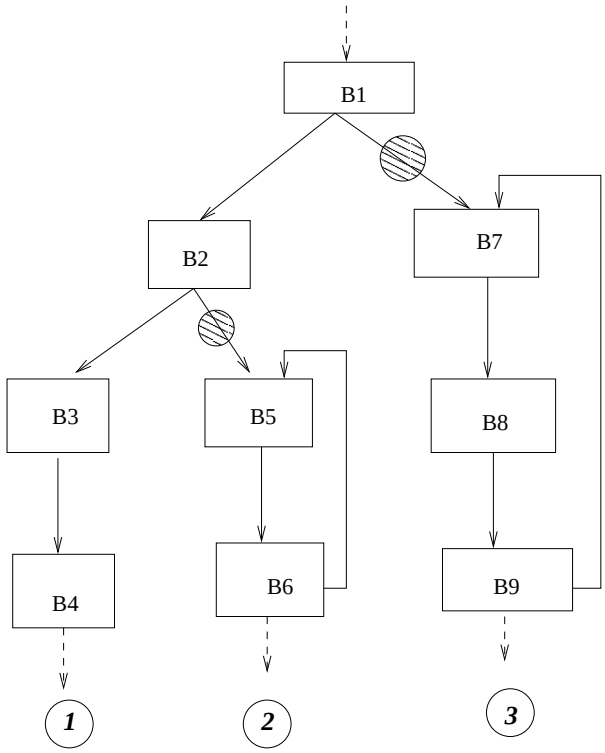

Figure 6. Placement of Frame Creation Instructions

Decimation in Time algorithm. Sierpinski and Fractal-Sq are two recursive functions which are used for fractal generation. Merge sort is one of the standard recursive algorithms for sorting. LD is a recursive program due to Ladner and Fischer which is used for parallel prefix computation [10].

We have run all the above example programs over the EMSIM simulator [17]. To be specific, we have used the EMSIM version 2.0 which simulates Linux-$2.4.18$ running over the Intel StrongARM-110 processor. The baseline of our benchmark studies is the code performance in relation to gcc 2.95.2 (GNU/Linux) cross compiler for the ARM processor, with optimization flags all ON. The EMSIM simulator captures both the system and the user call overheads. In this sense, it is a complete embedded system simulator. We have considered two parameters to compare the programs when they have been subjected to the LARS and when they have not. The two parameters are the the total number of cycles to execute the programs and the energy consumed (in milli Joules) by the processor to run the programs. Table 1 shows all such comparisons.

Observe from the table that for the call-intensive programs, especially in the case when the run time call tree rapidly grows, the energy consumption and performance

both improve from 26% upto 47%. This phenomena we see in case of Fibonacci, Ackerman, Tak, Sierpinski and Fractal-Squire functions. In the case of FFT, we observe 7% improvement both in performance and energy consumption. In case of Merge sort and LD function, such improvements are only around 2%. The reason is that such functions are not call-intensive and further, the computations inside the body of the functions dominate the cost of creating activation records.

We have also performed experiments over some benchmark programs for embedded systems; however their number is few and we are in the process of applying the LARS over a wide class of programs for embedded systems. For these experiments, we have used the LART system [11], which is a StrongARM based embedded computer system running Linux operating system. Table 2 shows the performance results for three programs from the PowerStone benchmark suite [12]. The program $jpeg.c$ is used for video data compression. The $g3fax.c$ is a FAX data encoder/decoder program. The program $pocsag.c$ is used for error correction in digital communication. Observe the 40% benefit in case of $jpeg.c$ in terms of energy consumption and running time. The reason is that this program makes a total number of 21000 calls out of which around 19000 calls are made to a single routine and this routine has been subjected to our LARS strategy. We even get better performance for the $g3fax.c$ program. The program $pocsag.c$ does not make a large number of calls to the routines which have been subjected to LARS; therefore the amount of saving is not much.

6 Discussion

6.1 LARS vs. related optimization strategies

The LARS is similar in spirit to Chow's *shrink wrapping*; however, there are significant differences. The Chow's strategy assumes that the available register set is partitioned into the subsets of caller-save and callee-save registers, and it is applied to optimize the placement of save/restore operations for the callee-save registers. The LARS does not assume any such division of the available register set. Even if there is a partition, the LARS extends the Chow's strategy in terms of scope and granularity; it tries to eliminate many memory references and it places the code of creating frames along control paths where they are needed.

The shrink-wrapping method is an intra-procedural approach, whereas the LARS can be viewed as an inter-procedural method, inter-procedural in the sense that the interface between two procedures is short-circuited. If A calls B, and a MICP in B does not require a frame, it may seem as if the MICP of B is executed in the environment of A.

The LARS also generalizes leaf-call optimization. The latter is applied to leaf level routines in the call graph of the program, whereas the former is applied to the leaf-level calls in the run time call tree (RTCT). As discussed in [14], an activation record is either not created (for some leaf-routines) or it is created in the procedure entry. Upgrading this mechanism, i.e. moving the creation of frames, to the individual MICPs is the novelty of the LARS. Furthermore, it should be obvious to see that the LARS subsumes the tail call optimization.

6.2 Significance of the LARS

1. Function calls are major bottlenecks for efficient register usage. They are more so in presence of recursion. Assuming that most of the interior nodes of a run time call tree have at least two children, which often is the case, LARS can remove the creation of activation records in 50% of the cases. As pointed out in the Ackerman Function example, LARS in combination with the tail–recursion optimization can be a powerful optimization strategy. Figure 7 shows the regions in the run time call tree (RTCT) for which activation records will not be created. The figure represents a RTCT in which region 1 shows the calls which are leaf-calls (functions which make no further calls in their definitions), and region 2 shows the calls which are non-leaf calls (functions which have calls in their definitions) but do not make calls in the particular run of the program. LARS avoids creation of frames for the calls within regions 1 and 2. There may be leaf-level calls in RTCT which may not be computed in registers alone (i.e. they require memory accesses) and therefore would require their frames. Region 3 shows such calls. RISC architectures in general, and RISC processors used for embedded systems in particular, usually have reasonably large number of registers [9]; they could be efficiently utilized to eliminate the creation of activation records for leaf–level calls in the RTCT.

2. LARS is an effective strategy in case of call-intensive programs, be they recursive or non-recursive. LARS can be an effective strategy for embedded systems from the viewpoint of energy consumption. It is true that recursion is not frequently used in programs for embedded systems. However, as we have observed in cases of $jpeg.c$ and $g3fax.c$, a program can be call-intensive without being recursive, and hence can be optimized though LARS. Embedded programs can be subjected to LARS in the following scenarios.

    - Most of the library calls, if not unfolded, can be subjected to LARS. Such library calls may not be leaf-level calls in the static call-graph, but

Benchmark Programs	Clock cycles ($\times 10^6$) (Unopt)	Clock Cycles ($\times 10^6$) (Opt)	saving in clock cycles	Energy Cons (in mJ) (Unopt)	Energy Cons (in mJ) (Opt)	% saving in energy
Fib(30)	1831.51	1322.95	27.7%	3199.35	2246.07	29.7%
Ack(3,6)	204.73	151.30	26%	359.08	260.258	27%
Tak(10,2,10)	16.76	8.82	47.4%	29.5	14.86	43%
FFT (128 samples)	.753	0.699	7.2%	1.34	1.24	7.6%
Sierpinski (gen = 9)	11.66	7.8	33%	20.21	13.19	34.7%
Fractal-Square	48.11	35.08	27%	82.3	60.11	27%
Merge Sort	5.25	5.13	2.2%	9.01	8.79	2.5%
LD fun (Prefix Sum)	3.56	3.47	2%	7.85	7.67	2.7%

Table 1. Results of some recursive benchmark programs

Benchmark Programs	Energy Consumed (in Joules) (Unopt)	Energy Consumed (in Joules) (Opt)	benefit	running time (mili sec.) (Unopt)	running time (mili sec.) (Opt)	benefit
jpeg.c	4.24	2.55	39.9%	1570.4	980.8	37.5%
g3fax.c	1.78	0.92	48.3%	741.7	340.7	54%
pocsag.c	0.64	0.60	6.25%	246.2	230.8	6.26%

Table 2. Results for some benchmark programs for Embedded Systems

they may have MICPs without involving function calls; i.e. such calls become leaf-level calls in the RTCT. And all such calls, in addition to the leaf-level calls in the call-graph, may possibly be executed without using activation records.

- In programs for embedded systems, function calls usually check a number of constraints, and when they do not hold, the functions exit. All such cases will give rise to MICPs which are likely to be executed without memory references, and hence activation records can be avoided.

3. Function unfolding is an effective strategy for eliminating function calls, and further they help in better register utilization [7]; however, they do increase the code size. Function unfolding is usually performed over leaf–routines. The LARS, wherever applicable, gives most of the benefits of *function unfolding* without actually unfolding the functions.

4. LARS is of particular importance to functional languages. Functional programs involve mostly function calls and the functions are usually small in size; therefore, the LARS can eliminate a sizable quantity of activation records.

5. The LARS is a static analysis technique. It can be performed after global register allocation and it does not require any additional infrastructure. Assuming that the CFG has been built and the global register allocation has been done, the LARS requires (a) to find the MICPs, and (b) serches over the basic blocks associated with the MICPs. And the complexity of both such tasks are linear in the number of instructions.

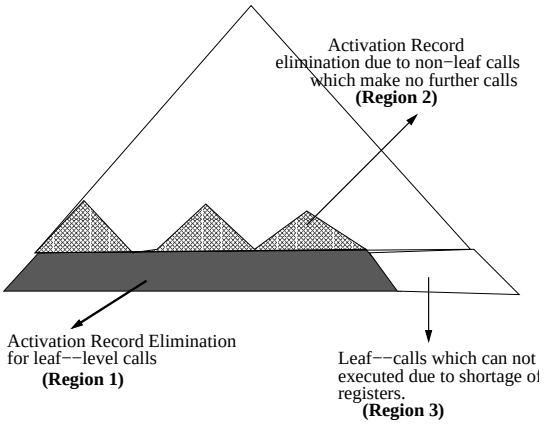

Figure 7. Subgraphs in the RTCT identified by categories

7 Conclusion

In this paper, we have discussed LARS, a new optimization strategy, which can eliminate the creation of activation records for recursive and non–recursive calls occurring at the bottom level(s) of the run time call tree. This strategy could be applied with ease to any kind of programs though its impact will be much more pronounced in case of functional programs. LARS tries to minimize memory accesses, and therefore it can be an effective strategy for lowering the power consumption for battery operated embedded systems. Though we have examined a small number of benchmark programs for embedded systems, the results indicate that LARS can be an effective strategy for embedded programs. We are now in the process of applying LARS to a wide spectrum of benchmark programs for embedded system. In addition, we intend to develop strategies so that LARS can be applicable to a wider variety of scenarios.

Acknowledgements

The idea of activation records being lazy came out from a discussion with A. Sanyal from I.I.T. Bombay. The first author had many useful discussions about this notion with S. Ramesh. Thanks to both of them.

References

[1] Allen R., Kennedy K., *Optimizing Compilers for Modern Architectures*, Morgan Kaufmann, 2002.

[2] *ARM: The Architecture for the Digital World*, Website: http://www.arm.com, 2002.

[3] Benini L., De Micheli G., System-Level Power Optimization: Techniques and Tools, ACM Transactions on Design Automation of Electronic Systems (TODAES), Vol. 5(2), April 2000, pp. 115–192.

[4] Chakrapani L.N., Korkmaz P., Mooney III V.J., Palem K.V., Puttaswamy K., Wong W.F., The Emerging Power Crisis in Embedded Processors: What can a poor compiler do? (Invited Talk), Proc. of the 2001 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES 2001), Atlanta, November 2001, ACM Press, pp. 176–180.

[5] Chow F., Minimizing Register Usage Penalty at Procedure Calls, ACM Conference on PLDI, 1988, 85–94.

[6] Chow F.C., Hennessy J.L., The Priority-Based Coloring Approach to Register Allocation, ACM Transactions on Programming Languages and Systems, Vol 12 (4), October 1990, pp. 501-536.

[7] Davidson J.W., Holler A.M., Subprogram Inlining: A Study of its effects on Program Execution Time, IEEE Tr. on Software Engineering, Vol 18(2), February 1992, pp. 89–102.

[8] Energy Simulator for StrongARM-Linux based Embedded System, Website http://www.ee.princeton.edu/ tktan/emsim/

[9] Hennessy J.L., Patterson D.A., *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, 1996.

[10] Ladner R.E., Fischer J., Parallel Prefix Computation, Journal of the ACM, Vol. 27(4), 1980, pp. 831–837.

[11] LART, website: *http://www.lart.tudelft.nl*

[12] Malik A., Moyer B., Cermak D., A Low level Unified Cache Architecture Providing Power and Performance Flexibility, International Symposium on Low Power Electronics and Design, June 2000.

[13] Morel E., Renvoise C., Global Optimization by Supression of Partial Redundancies, CACM, Vol 22(2), pp. 96–103.

[14] Muchnick S.S., *Advanced Compiler Design & Implementation*, Morgan Kaufmann, 1997.

[15] Satpathy M., Sanyal A., Venkatesh G., Improved Register Usage for Functional Programs through Multiple Function Versions, The Journal of Functional and Logic Programming, Volume 1998 (7), MIT Press, December 1998, pp. 1–46.

[16] Seal D., *ARM Architecture Reference Manual (2nd Edition)*, Addison Wesley, 2001.

[17] Tan T.K., Raghunathan A., Jha N.K., EMSIM: An Energy Simulation Framework for an Embedded System, Proc. ISCAS, 2002.