Compiler Directed Automatic Stack Trimming for Efficient Non-Volatile Processors

Qingan Li
Wuhan University
No. 299, Bayi Road, Wuchang
Wuhan, China
qingan@whu.edu.cn

Mengying Zhao
City University of Hong Kong
Tat Chee Ave
Kowloon, Hong Kong
my19900808@gmail.com

Jingtong Hu
Oklahoma State University
202 Engineering South
Stillwater, OK, 74078
jthu@okstate.edu

Yongpan Liu
Tsinghua University
Qinghua Yuan, Haidian District
Beijing, China
ypliu@tsinghua.edu.cn

Yanxiang He
Wuhan University
No. 299, Bayi Road, Wuchang
Wuhan, China
yxhe@whu.edu.cn

Chun Jason Xue
City University of Hong Kong
Tat Chee Ave
Kowloon, Hong Kong
jasonxue@cityu.edu.hk

ABSTRACT

Wearable devices are becoming increasingly important in our daily lives. Energy harvesting instead of battery is a better power source for these wearable devices due to many advantages. However, harvested energy is often unstable and program execution will be frequently interrupted. Non-volatile processors demonstrate promising advantages to back up volatile state before the system energy is depleted. But Non-volatile processors require additional memory for backing up, thus introducing non-negligible overhead in terms of energy, runtime as well as chip area. In this work, we target at non-volatile register reduction for energy harvesting based wearable devices. This paper proposes to stack trimming the memory footprint via a novel compiler directed method. The evaluation results deliver on average 28.6% reduction of non-volatile register files for backing up stack area, with ultra low runtime overhead.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*compilers, optimization, run-time environment*

General Terms

Design

Keywords

Stack, compiler, non-volatile processor

DAC '15 June 07 - 11, 2015, San Francisco, CA, USA
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3520-1/15/06 $15.00
http://dx.doi.org/10.1145/2744769.2744809 .

1. INTRODUCTION

In recent years, we have witnessed exponential growth of cyber-physical systems including wearable devices. Wearable technology refers to mobile electronic devices that are worn on a user's body or attached to their clothes. Wearable technology revolutionizes the quality of our daily lives, and is believed to become a $1.6 trillion business in the near-future [1]. In wearable devices such as health monitors, batteries are no longer favored due to size limitations, safety concerns, and recharging inconvenience. As an alternative, energy harvesting is proposed to power wearable cyber-physical devices.

Energy harvesting is a favorable power source since it can harvest energy from solar, electromagnetic radiation, or thermal sources to power electronics [13]. It provides better user experiences than batteries in an environment-friendly manner. However, energy harvesting power sources are intrinsically unstable [5]. Due to this instability, program execution may be frequently interrupted. With CMOS-based volatile processors, the intermediate results will be lost after power outages and thus result in vain re-executions, which imposes heavy energy overhead and even infeasibility to finish for large tasks. Non-volatile (NV) processors, in which non-volatile memory is attached to the volatile logics to back up program state information, have been developed recently to address this issue [15, 16]. Each time that there is a power outage, the processor's state will be saved to the non-volatile memory, which is known as checkpointing. Thus, the next time that the power comes back on, the processor's state is reloaded and program execution resumes.

However, the attached non-volatile memory induces considerable energy and runtime overheads, as well as occupies a significant amount of chip area. The program state typically consists of the data stored in the volatile memory and register files, where the memory content is commonly the major concern. The memory content typically consists of three separate parts: stack data, static data and heap data, among which stack data often holds a major share. In this work, we propose a compiler directed stack trimming strategy in order to reduce the program state size, which in turn reduces the required non-volatile memory size. We explore the opportunity of overlapping the caller function's frame and callee function's frames so that the corresponding address space can be shared. To the best of our knowledge, this is the first work to reduce NV register size via compilation based stack trimming. The proposed technique is orthogonal to hardware compression logics, therefore they can work together for better efficiency. Specially, this paper makes the following contributions:

- Propose a novel compiler directed stack trimming method with negligible overhead;
- Propose to apply the afore-mentioned stack trimming method to reduce the program state size for energy harvesting based systems;
- Conduct a set of experiments to evaluate the efficacy of the proposed strategy.

The remainder of this paper is organized as follows. Section 2 summaries related work. Section 3 introduces preliminaries and presents motivation examples. Section 4 presents the stack trimming scheme. Section 5 presents evaluations and Section 6 concludes this paper.

2. RELATED WORK

In this section, we will describe related work on energy harvesting systems and non-volatile processors.

Energy harvesting sources including solar, wind, finger motion and footfalls, are studied to be unstable [4, 2]. In order to overcome the instability, NV processors have been proposed for energy harvesting powered devices [15, 16]. Check-pointing has been shown to be an efficient method for saving the runtime state [8]. In NV processors, before energy runs out, the volatile execution states are checkpointed into the non-volatile memory. After the system is recharged, the execution states are copied back and thus the program can be resumed efficiently. FRAM is preferred as the attached non-volatile memory (NVM) due to its comparable access efficiency to SRAM and the good endurance of 10^{14} write cycles [16].

Considering the size limitation of wearable devices, the necessary NVM size for checkpointing should be minimized. To achieve this goal, data compression based hardware designs are proposed to reduce the content to back up [12, 14]. These techniques are hardware-based and the efficacy is highly dependent on compression ratios. In this work, we tackle this problem from the compiler perspective by stack trimming.

There are previous researches on stack size reduction. The work in [9] employs global function inlining to reduce maximum stack memory requirements. Some work proposes to reduce the stack size by keeping only one instance of a local variable in recursive programs if it is guaranteed that this local variable is at all recursive calls [10]. In this work, we conduct stack trimming from a different angle by manipulating frames of the caller and callee functions. The efficacy of the proposed scheme is confirmed by evaluations.

3. PRELIMINARIES AND MOTIVATION

In this section, background informations are introduced first, including the non-volatile processor, and the conventional stack allocation scheme. Then a motivation example is presented to illustrate how allocation schemes affect the size of NV processors.

3.1 Non-volatile processor

Fig. 1 shows the structure of the state-of-the-art NV processor, including on-chip non-volatile register files to back up volatile logics of the system when energy is depleted. Conventionally, all the volatile status, e.g. general purpose register, user variables and stack, are copied to the NV register file so that the program can resume after system is recharged. Thus the necessary NV register file size is conservatively designed to be the same as, or even larger than the volatile logic. Stack size dominates the internal volatile logics (around 63% [15]) and thus this paper focuses on the stack size reduction.

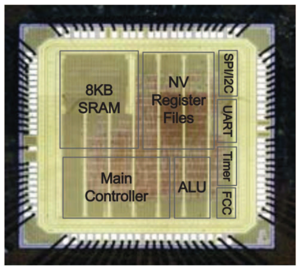

Figure 1: NV processor structure proposed in [15].

3.2 Live range

An object is alive at a program point if it may be referenced in future. All program points at which an object is alive constitute its live range. Live variable analysis is a common technique to compute the live range of each object[3]. A key observation is that, the storage space assigned to an object can be safely reused by another object, if these two objects have disjoint live ranges. Based on this observation, a lot of space reusing schemes have been proposed to reduce the required storage size, among which register allocation is one of the most famous techniques.

3.3 Stack allocation

Program data are conventionally stored in three separate areas: stack area, static area, and heap area, among which stack area often contributes significantly for embedded systems. Each function instance is associated with a *frame* (also called *active record*) to store the context information for this function[3]. Local data, including local variables and compilation temporary variables, are stored in this frame. A conventional stack based allocator works as follows:

1. A specific memory address is assigned to the *main* function's frame.
2. Upon a function is called, the callee function's frame is allocated on top of the caller function's frame.
3. Upon a function returns, the callee function's frame is deallocated from the top of caller function's frame.

The composition of the allocated frames constitutes the stack area. Traditionally, the stack space is separately allocated for the caller and callee functions, which is too conservative and may results in a large total size. In this work, we propose a novel stack allocation scheme for stack trimming through address space sharing.

```
 1 struct T
 2 {
 3 int _i;
 4 int _j;
 5 char _arr[10];
 6 };

 7 int copyT( struct T *t1, struct T *t2 )
 8 {
 9 int i;
10 int j;
11 modify(&i);
12 t1->_i = t2->_i + i;
13 modify(&j);
14 t1->_j = t2->_j + j;
15 strcpy(t1->_arr, t2->_arr);
16 return 0;
17 }
```

Figure 2: A example program. For simplicity, the code of function *modify* and *strcpy* is not listed here.

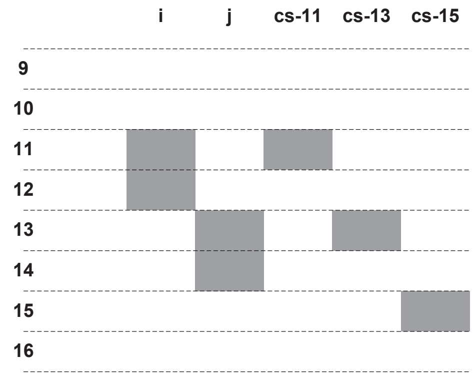

Figure 3: Live ranges for *i*, *j*, call site at line 11 (*cs-11*), call site at line 13 (*cs-13*), call site at line 15 (*cs-15*).

3.4 Motivation example

Here a motivation example is presented to illustrate how stack allocation schemes affect the stack size. The example code is shown in Figure 2. The *live range* for each local object of function *copyT* is shown in Figure 3. Note that here each call site is also viewed as a local object, and its size is equal to the callee function's frame size. Figure 4 illustrates the stack size under different stack allocation schemes. It is shown that the conventional allocation scheme, without any overlay in stack, holds the largest stack size, 20 bytes (Figure 4(a)). Since *i* and *j* have disjoint live ranges, they are overlaid in Figure 4(b). As a result, the frame size of *copyT* is reduced by 4 bytes, and the maximum stack size is in turn reduced to 16 bytes. Furthermore, in Figure 4(c), by overlaying call sites with disjoint live ranges, the maximum stack size can be reduced to 12 bytes.

From this example, we can conclude that objects with disjoint live ranges can share the same address without violating the data integrity, such as callee *cs-15* and caller *copyT*. Objects with overlapped live ranges need to be assigned with separate address like *i* and *cs-11*. In this work, we will apply the address sharing principle for objects with disjoint live ranges to achieve stack trimming.

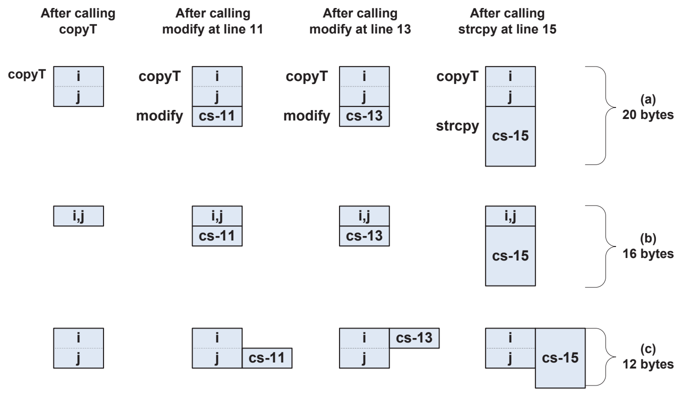
After calling copyT
After calling modify at line 11
After calling modify at line 13
After calling strcpy at line 15
copyT
modify
strcpy
i
j
i,j
cs-11
cs-13
cs-15
(a) 20 bytes
(b) 16 bytes
(c) 12 bytes

Figure 4: Comparison of stack size under different stack allocation schemes. Assume that the frame size of *copyT*, *modify* and *strcpy* is 8, 4 and 12 bytes, respectively. (a) The conventional allocation scheme. (b) The allocation scheme with data overlaying among data objects. (c) The allocation scheme with data overlaying among data objects and call sites.

4. COMPILER DIRECTED STACK TRIMMING

In this section, the compiler directed stack trimming technique is proposed. The challenges of the propose scheme lies in that the stack space should be shared as much as possible by objects, with the guarantee of data integrities as well as the function sequence in stack. Here, objects include variables which are not assigned to registers and all call sites. To achieve this goal, two steps are proposed as outlined below, which will be discussed in detail later.

1. **Function call graph construction**. The function call graph (FCG) is constructed to determine the order by which functions are processed.

2. **Data allocation based on FCG**. The local objects and call sites are allocated per function. With the help of FCG construction, the callee function is guaranteed to be processed before the caller function.

4.1 Function call graph construction

For a program, its function call graph (FCG) is a directed graph *FCG*=<*V*,*E*>, where *V* is the set of vertices and *E* is the set of edges. Each vertex represents a function. If a function *f* calls *g*, there is a directed edge from *g* to *f*.

If there is no recursive function calling in a program, its FCG is a directed acyclic graph (DAG). Then, a topological sorting can be conducted, where each element is a function, and a callee function always precedes the caller function.

If there are recursive function callings in this program, its FCG has cycles. This FCG needs to be transformed into a DAG by coalescing each cycle into one vertex. Then, a topological sorting can be conducted, where each element is either a function or a set of functions in the same cycle.

Algorithm 4.1 Data allocation based on FCG.

```
Input:
    DAG: FCG of the program with topological sorting;
    allocMap: an empty set of tuples (f,v,n), where f is a function,
    v is a local object or call site of f, and n is the offset address
    assigned to v.
Output:
    allocMap: the updated allocMap.
1: for each function f of in the FCG do
2:   if f is external, library or recursive function then
3:     skip;
4:   else
5:     conduct live variable analysis for f;
6:     StackColoring(f, allocMap);
7:   end if
8: end for
```

Each cycle is considered as one element, it always precedes its caller functions in the topological sorting.

4.2 Data allocation based on FCG

This step allocates local objects, as well as call sites, for each function. As illustrated in Algorithm 4.1, the stack allocation is conducted for each function in the order defined by the topological sorting of FCG. For each function, the live variable analysis is conducted first to determine whether a local object or call site can be placed overlapped with another; then data allocation is conducted to assign an offset address to each local object or call site. Note that external or library functions should be ignored, since their source code is unavailable.

Live variable analysis. Note that the traditional live variable analysis does not cover call sites. Therefore, a modification is needed here to compute the live range of each call site. In fact, the live range of the call site only includes the call instructions constituting this call site.

After the live analysis, a live range interference graph (LRIG) can be constructed. An *LRIG=<V,E>* is an undirected graph, where V is the set of vertices and E is the set of edges. Each vertex represents a local object. If an object has overlapped live range with another, there is an edge between these two objects. Live range analysis is conducted for each function to determine the live range of local objects and call sites.

Stack Coloring. With the LRIG constructed by live variable analysis, the data allocation problem can be viewed as a graph coloring problem: to find a coloring scheme to color the LRIG with the minimum number of colors under the constraint that, the neighbour vertices cannot be colored with the same color. Therefore, a heuristic graph coloring algorithm as illustrated in Algorithm 4.2 is proposed for data allocation. It always selects an object or call site with the largest size to allocate (line 7∼12). Note that, the object size can be determined at compilation time. Also, the call site size, which is the size of its callee function, can always be determined since the data allocation is conducted in the order of callee functions before caller functions. For each selected object, it always tries to assign a possibly lowest numbered color to it (line 16∼30). After stack coloring, it finalizes the data allocation by associating each color with an offset (line 36∼45). A special case is when the callee function is external, library or recursive, the call site size is set to be zero. Since it has the smallest size, this algorithm assigns the highest numbered color to it and then sets its offset with the

Algorithm 4.2 Stack coloring algorithm.

```
Input:
    f: the function to be processed;
    frameSizeMap: recording the frame size for each function.
Output:
    allocMap: the updated frameSizeMap.
 1: // Step I: stack coloring
 2: // colorMap is used to record the set of locals and call sites
    assigned for each color
 3: initialize colorMap to be empty;
 4: totalColors ← 0;
 5: while LRIG is not empty do
 6:   // select the object of largest size, then of highest degree
 7:   find the set of vertices with the largest size from f's LRIG;
 8:   if this set is not a singleton then
 9:     curObj ← a vertex with the highest degree from this set;
10:   else
11:     curObj ← the unique vertex in this set;
12:   end if
13:   // assign the possibly lowest numbered color to the selected
      object
14:   color ← 0;
15:   success ← false;
16:   while color <totalColors && !success do
17:     objs ← colorMap[color];
18:     if any obj in objs interferes with curObj then
19:       skip;
20:     else
21:       objs.add(curObj);
22:       success ← true;
23:     end if
24:     color ← color + 1;
25:   end while
26:   // need to assign a new color
27:   if !success then
28:     colorMap[totalColors].add(curObj);
29:     totalColors ← totalColors + 1;
30:   end if
31:   update LRIG by removing curObj;
32: end while
33: // Step II: data allocation according to stack coloring
34: color ← 0;
35: nextOffset ← 0;
36: while color <totalColors && !success do
37:   offset ← nextOffset;
38:   for each obj of colorMap[color] do
39:     obj.offset ← offset;
40:     if offset + obj.size > nextOffset then
41:       nextOffset ← offset + obj.size;
42:     end if
43:   end for
44:   color ← color + 1;
45: end while
```

caller function's frame size. As discussed later, it conforms to the conventional stack allocation scheme.

4.3 Implementation analysis

In conventional stack allocation, the callee function's frame is allocated on top of the caller's frame. This is implemented by inserting an instruction such as "*SUB ESP, n*" at the caller function's entry (before all call sites), where *ESP* is the stack pointer register and n is the caller function's frame size. And, an instruction such as "*ADD ESP, n*" is inserted at the caller function's exit to restore the stack pointer. Essentially, it implicitly assigns an offset equal to the caller function's frame size to all call sites.

The technique proposed in this paper makes a change that it assigns different offsets to different call sites. It can be

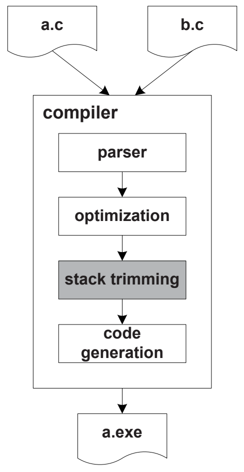

Figure 5: Experimental setup.

implemented by inserting "*SUB ESP, m*" before each call site individually where m is the offset assigned to this call site. And, after each all site, "*ADD ESP, m*" is inserted to restore the stack pointer. The overhead comes from the inserted stack pointer modification instructions, which is linear to the number of call sites.

For the special cases when the callee function is external, library or recursive, as stated in Section 4.2, its offset equates the caller function's frame size, which conforms to the conventional stack allocation. Therefore, there is completely no compatibility problem with legacy code.

5. EXPERMENTS

In this section, the experimental setup is introduced first, and then the experimental results are presented.

5.1 Experimental setup

To evaluate the proposed stack trimming method, we use the experimental setup as illustrated in Figure 5. The stack trimming method is implemented based on LLVM compiler infrastructure[6]. It works after the register allocation pass, since the stack allocation should collect and allocate the variables spilled out by the register allocation pass. We run the generated code with the PIN tool[7] to collect the stack size statistics. The benchmarks are from the powerstone suite [11].

In the experiments, three methods are evaluated. The *baseline* method applies the conventional stack allocation scheme. The *data overlay* method reduces the frame size of a function via assigning the same address to objects with disjoint live ranges. The *aggressive overlay* method is similar to *data overlay*, except that it reduces the total stack size by sharing the same address among the callee function's frames and the caller function's local objects.

5.2 Comparison of maximum stack size

The requirements of stack size are critical for energy harvesting NV processors. This is because with a smaller stack size, less program states are needed to back up before power failure, and thus the size of NV memory can be reduced. Also, the reserved energy budget for back up can also be reduced. As Figure 6 shows, for all benchmarks, the *aggressive overlay* method reduces more stack size than the *data overlay* method, and the *data overlay* method reduces the stack size compared to the *baseline* method. Compared to the *data overlay* method, although the *aggressive overlay* method cannot improve the frame size reduction, it reduces the total stack size by overlaying adjacent function frames. On average, the *data overlay* and *aggressive overlay* methods can reduce the stack size by 17.7% and 28.6% respectively, compared the *baseline* method.

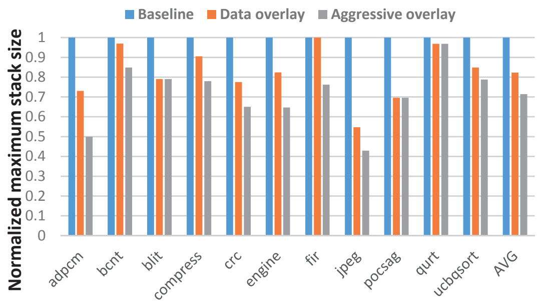

Figure 6: Comparison of maximum stack size.

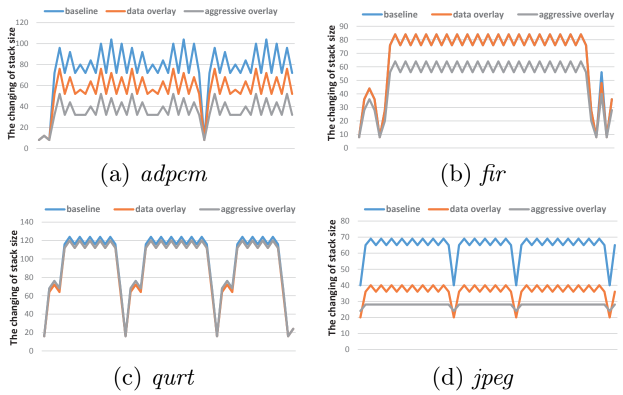

Figure 7: The first 50 changes of stack size (Byte).

5.3 Changes of stack size at call sites

During a program's run time, its stack size only changes at call sites. When the call site invokes the callee function, the callee function's frame is allocated on stack and the stack size increases. When the callee function returns to the call site, its frame is deallocated and the stack size decreases. Figure 7 shows the first 50 changes of stack size for several benchmarks. For *adpcm*, the *aggressive overlay* method is constantly better than the *data overlay* method, and the *data overlay* method is constantly better than the *baseline* method. Therefore, whenever the energy outage occurs, with the proposed stack trimming method, a smaller number of back up registers are needed. This phenomenon holds true for most of the benchmarks, and during most of a program's run time. But there are several exceptions.

As illustrated in Figure 7(b), for *fir*, the *data overlay* method cannot reduce the stack size, while the *aggressive overlay* method can significantly reduce the stack size, compared to *baseline*. This is because, the live ranges of local variables interfere with each other, while some of them are disjoint from live ranges of the call sites.

Benchmark(#)	adpcm	bcnt	blit	compress	crc	engine	fir	jpeg	pocsag	qurt	ucbqsort
# of call sites	43	10	12	27	3	15	20	12	34	14	15
# of instructions	1572	700	815	1608	184	839	785	597	1152	742	991
dyn. calling count	753	2	4	938	259	5229	346	31229	515	26	9142
dyn. instr. count	92360	56375	3476	222580	61937	651897	23601	7203505	93512	1853	335098

Table 1: Benchmark characteristics.

Reversely, for *blit*, *pocsag* and *qurt*, the *aggressive overlay* method cannot reduce the stack size more than the *data overlay* method, as illustrated in Figure 7(c). For these benchmarks, the live ranges of call sites always interfere with the live ranges of local objects.

Another exception is *jpeg*. As illustrated in Figure 7(d), the first function, *main*, has a larger frame size in *aggressive overlay* method than in the *data overlay* method. This is because that, during data overlaying process, the former method considers not only local variables, but also the callee functions' frames. In order to make the most of data overlaying, the local variables may be allocated with a larger space. However, this phenomenon occurs only close to the entry or exit of a program. This is because, when multiple function frames are allocated on stack, the former method can reduce the stack size by overlaying the adjacent frames.

5.4 Overhead analysis

The overhead of the proposed stack trimming method can be analyzed from two aspects: code size and run time overhead. As stated in Section 4.3, the implementation needs to insert *ESP* modification instructions before (after) the concerned call site for frame allocation (deallocation). The number of such instructions is no more than twice of the number of call sites. As shown in Table 1, compared to the total number of instructions, this code overhead is negligible. On the other hand, the run time overhead is determined by the execution count of the inserted instructions, which is no more than twice of the dynamic function callings. As shown in Table 1, compared to the total execution count of instructions, this run time overhead is also negligible.

6. CONCLUSION

Energy harvesting is a favourite power source for wearing devices. To overcome the unstable of energy harvesting, non-volatile memory based back up processors have been proposed to back up the program state before power failure. It is critical to reduce the size of program state information for frequent back up. This paper proposes a novel compiler directed stack trimming strategy to reduce the stack size. The experimental results shown that the proposed method can effectively reduce the stack size with negligible overhead.

7. ACKNOWLEDGMENTS

This work was supported partially by the NSFC (No.91118003, 61373039, 61170022, 61271269), High-Tech Research and Development (863) Program (No.2013AA01320), the Specialized Research Fund for the Doctoral Program of Higher Education (No.20130141110025l2), the Importation and Development of High-Caliber Talents Project of Beijing Municipal Institutions (No.YETP0102), and NSF CNS-1464429.

8. REFERENCES

[1] Morgan Stanley: Wearable Devices A Potential $1.6 Trillion Business: `http://www.wearabledevices.com/2014/11/20/morgan-stanley-wearable-devices/`.

[2] *19th Asia and South Pacific Design Automation Conference, ASP-DAC 2014, Singapore, January 20-23, 2014*. IEEE, 2014.

[3] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Princiles, Techniques, and Tools*. Addison-Wesley, 1986.

[4] X. Jiang, J. Polastre, and D. Culler. Perpetual environmentally powered sensor networks. In *Information Processing in Sensor Networks, 2005. IPSN 2005. Fourth International Symposium on*, pages 463–468, April 2005.

[5] A. Kansal, J. Hsu, S. Zahedi, and M. B. Srivastava. Power management in energy harvesting sensor networks. *ACM Trans. Embed. Comput. Syst.*, 6(4), Sept. 2007.

[6] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[7] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[8] B. Ransford, S. S. Clark, M. Salajegheh, and K. Fu. Getting things done on computational RFIDs with energy-aware checkpointing and voltage-aware scheduling. In *HotPower*, 2008.

[9] J. Regehr, A. Reid, and K. Webb. Eliminating stack overflow by abstract interpretation. In *Embedded Software*, pages 306–322. Springer Berlin Heidelberg, 2003.

[10] S. Schaeckeler and W. Shang. Stack size reduction of recursive programs. In *CASES*, pages 48–52, 2007.

[11] J. Scott and et al. Designing the low-power m*core architecture. In *Proceedings of IEEE POWER DRIVEN MICROARCHITECTURE WORKSHOP, 1998.*, 1998.

[12] X. Sheng, Y. Wang, Y. Liu, and H. Yang. Spac: A segment-based parallel compression for backup acceleration in nonvolatile processors. In *DATE*, pages 865–868, 2013.

[13] S. Sudevalayam and P. Kulkarni. Energy harvesting sensor nodes: Survey and implications. *IEEE Communications Surveys Tutorials*, 13(3):443–461, March 2011.

[14] Y. Wang, Y. Liu, S. Li, X. Sheng, D. Zhang, M.-F. Chiang, B. Sai, X. Hu, and H. Yang. Pacc: A parallel compare and compress codec for area reduction in nonvolatile processors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22(7):1491–1505, July 2014.

[15] Y. Wang, Y. Liu, S. Li, D. Zhang, B. Zhao, M.-F. Chiang, Y. Yan, B. Sai, and H. Yang. A 3us wake-up time nonvolatile processor based on ferroelectric flip-flops. In *ESSCIRC*, pages 149–152, Sept 2012.

[16] M. Zwerg, A. Baumann, R. Kuhn, M. Arnold, R. Nerlich, M. Herzog, R. Ledwa, C. Sichert, V. Rzehak, P. Thanigai, and B. Eversmann. An 82ua/mhz microcontroller with embedded FeRAM for energy-harvesting applications. In *ISSCC*, pages 334–336, Feb 2011.