Demand Code Paging for NAND Flash in MMU-less Embedded Systems

José A. Baiocchi and Bruce R. Childers

Department of Computer Science, University of Pittsburgh

Pittsburgh, Pennsylvania 15260 USA

Email: {baiocchi,childers}@cs.pitt.edu

Abstract—**NAND flash is preferred for code and data storage in embedded devices due to its high density and low cost. However, NAND flash requires code to be copied to main memory for execution. In inexpensive devices without hardware memory management, full shadowing of an application binary is commonly used to load the program. This approach can lead to a high initial application start-up latency and poor amortization of copy overhead. To overcome these problems, we describe a software-only demand-paging approach that incrementally copies code to memory with a dynamic binary translator (DBT). This approach does not require hardware or operating system support. With careful management, a savings can be achieved in total code footprint, which can offset the size of data structures used by DBT. For applications that cannot amortize full shadowing cost, our approach can reduce start-up latency by 50% or more, and improve performance by 11% on average.**

I. INTRODUCTION

In many embedded devices, application binaries are kept in NAND flash storage, which has relatively high access latency and energy compared to main memory. To be executed, an application binary must be loaded to the device's main memory [1]. In low-cost devices without a hardware memory management unit (MMU), copying the entire binary to main memory, known as *full shadowing*, is often used. In this approach, an application may suffer increased run-time and start-up delay. The application must execute long enough relative to its binary size that the copying cost from the slow flash device is amortized. The application boot-up delay can also be large, depending on code size, because the entire shadowing cost is paid upfront before execution begins.

Demand paging has been proposed as an alternative that allows partially loaded programs to be executed [2], [3]. It divides an application binary into equal-sized portions, called *pages*, which are copied from flash to main memory only when needed for execution. An MMU is typically used to generate a *page fault* when an unavailable page (i.e., in memory) is accessed. The page fault is handled in the operating system (OS) by loading the page. Demand paging reduces boot time and will typically help performance by loading only some pages, but it requires the complexity and cost of a processor with an MMU and a full OS that supports paging. A cost-sensitive device may not be able to justify these changes, particularly when the device would otherwise use a simple microcontroller and lightweight OS executive.

In this paper, we present a software-only approach for demand paging of code stored in NAND flash without requiring an MMU or OS support. The approach aims to improve boot-time and performance of programs that do not amortize their shadowing cost. The basis of the approach is *dynamic binary translation* (DBT), a technology that enables the control and modification of a program as it executes. A novel demand-paging service is added to DBT to load code along newly taken execution paths. Because DBT can introduce memory overhead, our demand-paging service is designed to reduce total code footprint. The code footprint savings can be used to help offset memory requirements from data structures maintained for DBT.

DBT provides important services in general-purpose systems, including virtualization, security, and dynamic optimization. Due to these uses, it has also gained attention for embedded systems, with much research on memory and performance efficient DBT for embedded processors [4], [5], [6]. DBT has also been used to create embedded software services [7], [8]. Our demand-paging service may be combined with these other services to increase their applicability and benefit in cost-sensitive devices.

This paper makes the contributions: (1) A design and evaluation of a DBT-based demand-paging service for code stored in NAND Flash, showing how boot time and overall execution time are improved. (2) The integration and unified management of the software buffers used by our DBT system to store original (untranslated) code and translated code. (3) An evaluation of our demand-paging service with different page replacement algorithms (FIFO, LRU) and different unified code buffer sizes, showing how code memory consumption can be reduced with little performance loss.

II. LOADING AN APPLICATION FROM NAND FLASH

Flash has become standard to store code and data files in embedded devices, due to its non-volatility, reliability and low-power consumption. NAND flash is common for storage because it has high density at a low cost per byte. A NAND flash chip is divided into multiple *blocks* and each block is divided into *pages*. In small chips, a page holds 512 bytes of data and 16 control bytes. NAND flash can only be read or written one page at a time. An entire block must be erased before its pages can be written. Reading data takes

tens of microseconds for the first access to a page, and tens of nanoseconds per each additional byte read. Flash has to be managed due to wear-out from write operations, which necessitates a *Flash Translation Layer* (FTL)[9]. The OS accesses flash as a block device. Since efficient random access to bare NAND flash is not available, application code stored in flash must be copied to main memory to be executed. There are two approaches to enable code execution: full shadowing and demand paging.

Full shadowing copies the entire contents of a program's binary from flash to main memory [10]. This approach is feasible when the binary fits in the available main memory – i.e., it leaves room for the program's data (stack and heap). However, as the size of the binary increases, the application's boot time and memory demand also increase. Further, the load time of the application may not be fully amortized because a program may execute for a short duration relative to the latency to load the application binary image.

Demand paging allows the execution of programs by partially loading a binary image as needed. With demand paging, load latency can be improved and memory requirements reduced by dividing code and data in NAND flash into logical *pages* and copying them to main memory when requested for execution. Because only the code that is executed is read from the slow flash device, there is less total load overhead, which can be more easily amortized. However, this approach requires an MMU and OS support to generate and handle a fault when a memory operation accesses a page that is not in main memory [2]. OS management also incurs overhead beyond the latency to access the flash device.

Ideally, a low-cost embedded device should get the advantages of demand paging (low load cost, quick start-up) without its complexity (MMU and OS paging). A software-only approach with a compiler and a run-time system is one way to achieve this goal [3]. A compiler technique, however, requires a program to be prepared ahead-of-time for execution. For devices with different memory organizations, separate versions of the software must be prepared, which is burdensome and complicates in-the-field updates.

Instead of a compiler, we use DBT to control code execution and provide demand paging for code in flash. Our technique complements full shadowing since it targets applications that have large start-up delay and/or cannot amortize shadowing cost. When full shadowing works well, we do not enable demand paging. This strategy needs no hardware or OS support, and it permits applications, including legacy ones, to be executed without advanced preparation.

III. DEMAND PAGING WITH DBT

Conventionally, DBT accesses an application's code from its process image in main memory [11], and the OS and hardware provide virtual memory and paging. In our approach, DBT takes the place of the OS loader. DBT controls application execution and accesses a binary image in flash to load code pages when needed for translation. DBT is incorporated as part of the system executive and instantiated one time to serve all
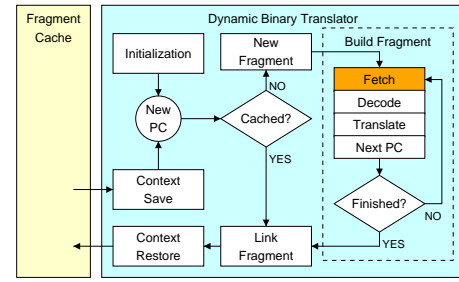

Fig. 1.   DBT system overview

applications that are incrementally loaded. The DBT system is copied, with the executive, from ROM to main memory on system boot-up.

A. Dynamic binary translation

DBT loads, examines and possibly modifies every instruction before it is executed for the first time. A high-level view of DBT is shown in Figure 1. To ensure that all untranslated application instructions are processed prior to their execution, the translator must be invoked whenever new application code is requested. New code is requested at the beginning of execution and every time a control transfer instruction (CTI) targets an untranslated application address.

As shown in Figure 1, the translator is first initialized and takes control of the application. The translator checks whether there is translated code for the current program counter (PC). If there is translated code for the PC, that code is executed. Otherwise, a group of translated instructions, called a *fragment*, is created, saved in a software-managed buffer, and executed. The software buffer is called the *fragment cache* (F$), as shown on the left of Figure 1.

To build a fragment, each instruction is fetched, decoded and translated for a particular purpose (e.g., to add security checks), as shown in the loop on the right of Figure 1. When incremental loading is applied by itself, the translate step simply copies most untranslated instructions to the F$. Translation stops when a CTI is found [5]. After a new fragment is built, application context is restored and the new fragment is executed. The translator replaces the untranslated target of a CTI with an exit stub *trampoline*. A trampoline "re-enters" the translator to find or create a fragment. A *context switch* is done when the translator is re-entered.

To avoid unnecessary context switches, the *fragment linker* overwrites each trampoline with a direct jump to its corresponding target fragment once the target is translated. Indirect CTIs may change their target at run-time, so they are replaced by a portion of code that maps the original application address targeted by the indirect CTI to its corresponding translated address. This code performs a lookup into an *indirect branch translation cache* (IBTC).

B. Adding Demand Paging to DBT

An application binary contains code and statically initialized data (e.g., literal strings). The binary also contains metadata that indicates where code and data should be placed in memory (i.e., their memory addresses). When full shadowing is used,
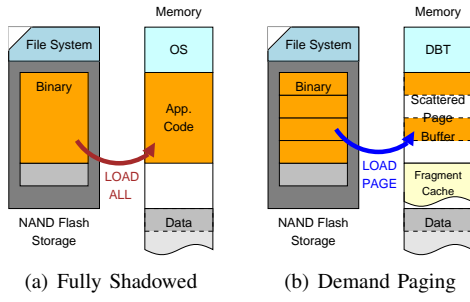
(a) Fully Shadowed   (b) Demand Paging

Fig. 2. Application Binary Loading

(1) Load & Translate   (2) Overflow: Evict Page

Fig. 3. Unified Code Buffer

the OS executive loads the code and data from the binary to main memory and starts application execution, as illustrated in Figure 2(a).

Our approach is shown in Figure 2(b); DBT replaces the OS loader. During initialization, the DBT system loads data from flash to main memory. Code pages are loaded on-demand when the translator builds a new fragment. A memory buffer, called the *page buffer*, holds the code pages copied from the binary image. To achieve the best performance, the page buffer can be made as big as the code segment in the binary. Each loaded page is then placed at the same memory address that a full shadowing loader would put it. We call this approach the *scattered page buffer* (SPB).

To perform demand paging, the *fetch* step in Figure 1 has to be changed to detect and handle page faults in software. The new fetch step implements Algorithm 1.

Algorithm 1 Fetch step with scattered page buffer

1: $instruction \leftarrow *(PC)$
2: **if** $instruction = 0$ **then** {software page fault}
3:      $page \leftarrow (PC - TADDR)/PSIZE$
4:      lseek(bfd, $TOFFSET + page * PSIZE$, 0)
5:      read(bfd, $TADDR + page * PSIZE$, $PSIZE$)
6:      $instruction \leftarrow *(PC)$
7: **end if**

In Algorithm 1, *TADDR* is the start address of the code segment in main memory, *TOFFSET* is the start offset of the code section in the binary, and *PSIZE* is the size of a page. The algorithm assumes the SPB is initialized to zeroes during allocation and that 0 is not a valid encoding for an instruction (a different sentinel value may be used).

The fetch step tries to obtain an instruction needed by the program so it can be decoded and translated. Line 1 reads the instruction from PC address. If the instruction is present, the only additional cost relative to translating full shadowed code is checking for a page fault on line 2. When a page fault is detected, a page number is computed for the faulty address and the necessary code page is copied from the binary to its corresponding address in main memory (lines 3-5). On line 6, the instruction is read again after the page is copied.

The SPB requires enough physical memory to hold untranslated code pages, translated code and data. Demand paging with DBT improves the application's boot time since application execution starts just after copying any data segments, reading the first page and forming the first fragment. Only pages containing executed code are loaded. Full shadowing
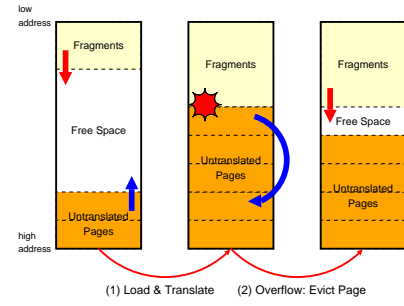
may load pages that are never executed. Loading only the pages containing executed code reduces total load time, which helps amortize time spent on translation.

C. Unified Code Buffer

Executing code under DBT increases memory usage since the original code pages are treated as data and the application code that is actually executed is held in the fragment cache. To mitigate the increase in memory pressure due to DBT, we combine a page buffer (PB) and the F$ into a *Unified Code Buffer* (UCB). The UCB holds both untranslated pages and translated fragments. Its size can be restricted to be smaller than the original binary size in flash. Thus, the code memory footprint can be made *smaller* with our demand-paging service than with full shadowing, and the savings in code footprint can be used to offset the relatively small data structures needed by DBT [6], [12].

Figure 3 illustrates the organization and management of the UCB. The UCB has a fragment portion that starts at its lowest address and grows towards its highest address. There is an untranslated code page portion at the highest address that grows toward the lowest address. The first loaded page is placed at the bottom and each new page is placed on top of the previously loaded page as long as there is empty space in the UCB. When the UCB is full and a new page must be loaded, a *page replacement algorithm* chooses a cached page to be replaced. When the fragment portion of a full UCB needs to grow, the replacement algorithm selects a page to overwrite with the code page at the top of the page region. Then, the space used by the top page is assigned to the fragment portion of the UCB. A full UCB may need to be repartitioned frequently if the pages are small, with a negative impact on performance. To avoid frequent UCB management, multiple pages can be removed at once from the page region and assigned to the fragment region.

Algorithm 2 Fetch step with unified code buffer

1: $page \leftarrow (PC - TADDR)/PSIZE$
2: $offset \leftarrow (PC - TADDR) \bmod PSIZE$
3: **if** $pmap[page] = 0$ **then** {software page fault}
4:      lseek(bfd, $TOFFSET + page * PSIZE$, 0)
5:      $pmap[page] \leftarrow$ getpframe()
6:      read(bfd, $pmap[page]$, $PSIZE$)
7: **end if**
8: $instruction \leftarrow *(pmap[page] + offset)$

TABLE I
SIMPLESCALAR CONFIGURATION

Parameter	Configuration
Fetch queue	4 entries
Branch predictor	bimodal, 4 cycle mispred. penalty
Branch target buffer	128 entries, direct-mapped
Return stack	3 entries
Fetch/decode width	1 instr./cycle
Issue	1 instr./cycle, in-order
Functional units	1 IALU, 1 IMULT, 1 FPALU, 1 FPMULT
RUU capacity	8 entries
Issue/commit width	2 instr./cycle
Load/store queue	4 entries
L1 D-cache	16 KB, 4-way, FIFO, 1 cycle
L1 I-cache	16 KB, 4-way, FIFO, 1 cycle
Bus width	4 bytes
Memory latency	36 cycles first, 4 cycles rest
Flash page size	512 bytes
Flash read latency	325,000 cycles

TABLE II
NAND FLASH PAGES READ (512 BYTES/PAGE)

Program	FS	SPB	Program	FS	SPB
adpcm.dec	81	53	patricia	116	110
adpcm.enc	81	53	pgp.dec	524	318
basicmath	103	101	pgp.enc	524	290
bitcount	86	62	qsort	113	79
blowfish.dec	98	55	rijndael.dec	152	102
blowfish.enc	98	55	rijndael.enc	152	103
crc	83	58	rsynth	243	192
dijkstra	110	73	sha	84	57
fft	92	80	stringsearch	115	79
fft.inv	92	81	susan.cor	149	88
ghostscript	2047	971	susan.edg	149	95
gsm.dec	185	122	susan.smo	149	82
gsm.enc	185	142	tiff2bw	509	374
ispell	236	164	tiff2rgba	570	375
jpeg.dec	277	168	tiffdither	507	397
jpeg.enc	253	161	tiffmedian	517	368
lame	470	391	typeset	1230	909

Fig. 4. Boot time with DBT/SPB relative to DE/FS

When using the UCB, fetch invokes Algorithm 2. Lines 1-2 compute the page number for the PC address and the offset of the instruction within that page. A map table (*pmap*) holds the address where each page has been loaded. Line 3 checks whether *pmap* has the page number. If not, the page fault is handled on lines 4-6. On line 8, the instruction is read.

`getpframe` (line 5) finds a free page frame in the PB for the new page. When the PB is full, one currently loaded page must be replaced. A *page replacement algorithm* chooses which page to replace. We consider two standard replacement algorithms: FIFO (first-in, first-out) and LRU (least recently used). Unlike hardware demand paging, LRU is defined in terms of memory accesses done for translation rather than memory accesses for execution. Since it is difficult to ensure that a page is no longer needed, performance may degrade if a replaced page is loaded again.

IV. EVALUATION

A. Methodology

We implemented our techniques in the Strata DBT [11] for SimpleScalar/PISA. For direct execution with full shadowing, we use Strata to load the application binary into main memory and then transfer control to the original code (no translation is done). We enhanced SimpleScalar's timing simulator to obtain boot time, total execution time and number of flash page reads. Table I shows the simulator configuration, which resembles an 400 MHz ARM1176JZ(F)-S processor.

SimpleScalar uses the host OS to emulate system calls. We modified its system call interface to read binary instructions from a simulated flash device. According to experiments [13], the bandwidth for reading a 512-byte NAND flash page from a Kingston 1GB CompactFlash card is 0.6 MB/s, regardless of access pattern. Thus, we add 325,000 cycles to the simulator's cycle count for every page read from flash. We use MiBench with large inputs. We could not compile `mad` and `sphinx`.

B. Scattered Page Buffer Results

Demand paging with SPB *reduces the number of flash page reads* by 31% average versus full shadowing (FS) as shown in Table II. The page count includes data and code pages. Some programs have a large benefit; e.g., *ghostscript* has a 53% reduction (from 2047 to 971 page reads). This program can produce its output in several formats, but only one format is requested in a single execution, so many of its code pages are never accessed. However, applications like *basicmath* and *patricia*, have a small reduction (2% and 5%) because most of their pages are executed.

DBT with SPB also reduces boot-up latency until the first application instruction is executed by at least 50%. This benefit is shown in Figure 4, which lists percentage reduction in boot latency for DBT with SPB (DBT/SPB) versus direct execution with full shadowing (DE/FS). *tiff2bw* and *tiffdither* have the smallest reductions, around 50%. *typeset* and *ghostscript* have impressive reductions; their new boot time is just 13.5% and 14.4% of their original time!

To obtain good performance with DBT, the time spent in translation must be amortized by a reduction in other parts of total execution time. DBT overhead has to be less than or equal to the savings obtained by avoiding flash reads to achieve a benefit from the service. Programs that cannot amortize the load cost from full shadowing will benefit the most, while programs that successfully amortize shadowing cost will likely suffer slowdown with the technique.

Figure 5 shows speedup for demand paging. Speedup is relative to full shadowing and native execution without DBT. Execution time includes all latency cost for loading, translating (for DBT only) and executing application code. The figure shows two benchmark groups: ones that amortize shadowing cost (left) and ones that do not (right). The overall average speedup is 1.03 (bar "avg-all"). Many programs have better or similar performance with DBT/SPB than DE/FS. The highest speedups (right side) are for *pgp.encode* (1.41x), *jpeg.decode*
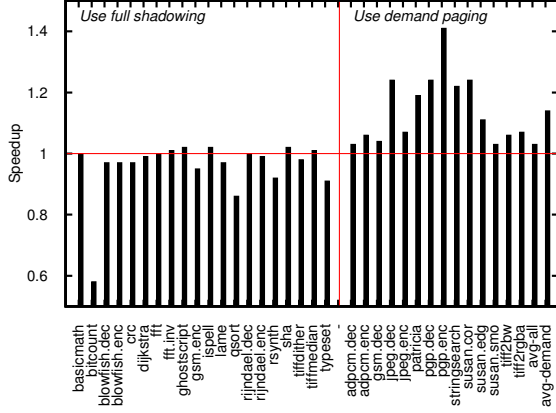
Fig. 5. Speedup with DBT/SPB relative to DE/FS

(1.24x), *pgp.decode* (1.24x) and *susan.corners* (1.24x). These programs execute for short periods relative to the number of flash reads with full shadowing.

Several programs can amortize full shadowing cost (left side), which causes a loss (or a relatively small gain) for demand paging. *bitcount* suffers a large loss with a 1.72 slowdown. This program is small, touches most pages, and executes for a long time given its size. In this situation, demand paging is not needed since it is ideal for shadowing. There is overhead in other programs (e.g., *qsort* and *typeset*) for this reason but it isn't nearly as high (16% and 10%).

Programs that do not need demand paging can be identified with offline or online profiling. Offline profiling can find the ratio of execution to full shadowing cost. If this ratio is favorable for full shadowing, then metadata can be used to tell the DBT system to load the full image. Alternatively, demand paging can be used upfront with online profiling of the ratio of load activity to total execution. If the ratio shows that demand paging is unnecessary, DBT could load any missing pages and bail-out from further paging.

When demand paging is disabled for programs that don't need it (left side of figure), the speedup of DBT/SPB is 1.03-1.41, with a 1.14 average (bar "avg-demand"). These results indicate that demand-paging provided by DBT can be more effective in MMU-less embedded systems than direct execution with full shadowing, particularly when demand paging can be selectively enabled/disabled on a per-benchmark basis. It always reduces boot time for MiBench and improves overall performance in most cases.

C. Unified Code Buffer Results

Both the SPB and F$ consume memory; however, combining them into a unified code buffer (UCB) allows control of code memory consumption. We studied the performance effect of enabling the UCB and limiting its size. This may allow placing the UCB in a small on-chip SRAM, if available [8], [5], or offsetting the memory cost of the data structures needed by the DBT.

Compared to an SPB with an unlimited F$, the UCB adds two sources of overhead: additional flash reads due to

premature page replacements and UCB repartitioning. We set the size of the UCB to a percentage of the code section's size in the application binary. To avoid frequent repartitioning, the amount of memory added to the fragment region on each UCB repartitioning is set to 5% of the code size.

We first determine which page replacement algorithm works best for the UCB. LRU has a higher management cost than FIFO, since the page replacement order has to be updated not only when the page is loaded, but also every time the translator accesses it. However, this cost may be worth paying since LRU can lead to better replacement decisions. Table III shows the number of flash reads made with both algorithms when the size of the UCB is 75% of the code size with full shadowing. The results indicate that LRU is often better than FIFO. FIFO has fewer flash reads in only 7 programs, while LRU has fewer flash reads in 20 programs. Thus, we use LRU.

The final experiment in varies UCB size as shown in Figure 6, which gives UCB speedup relative to direct execution with full shadowing. SPB results are included for comparison. The UCB size limits are 175%, 75% and 50% of the code size with full shadowing. With a 175% limit, the programs need no more flash reads than SPB. With a 50% limit, some programs did not run to completion because the available memory is too small for growing the fragment region (e.g., *basicmath*). For this reason, we do not consider UCB-50% further.

As UCB size is decreased, the speedup is lower since more flash reads are done to reload evicted pages. For example, in *pgp.dec*, the 1.24x speedup with the SPB is reduced to 1.2 (UCB-175%) and 1.18 (UCB-75%). A similar trend happens in *jpeg.decode*, where speedup goes from 1.24x with SPB to 1.19x (UCB-175%) and 1.14x (UCB-75%).

With UCB-75%, 25% of the memory space that would be consumed by full shadowing can be "spent" for other purposes in DBT-based demand paging. This provides one word for every three instructions in the UCB to hold DBT data structures (i.e., mapping tables). The size of the mapping tables depends on the number of instructions in the UCB, and a 1:3 ratio is sufficient, particularly when data and code footprint reduction for DBT are applied [12], [6], [8], [5], [14]. Overall, the average speedups (bars "avg-all") are 1.02x with UCB-175% and 1.01x with UCB-75%. When DBT-based paging
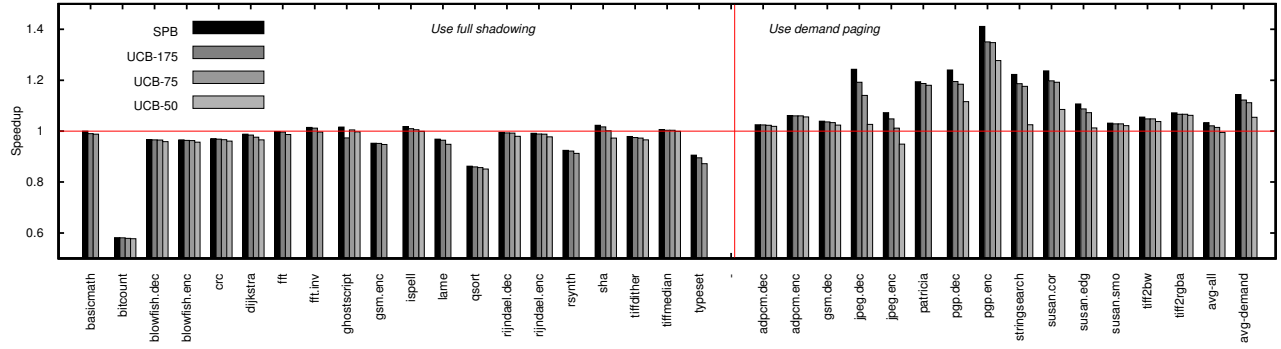
TABLE III
NAND FLASH PAGES READ WITH UCB-75%

Program	FIFO	LRU	Program	FIFO	LRU
adpcm.dec	56	55	patricia	153	154
adpcm.enc	58	54	pgp.dec	329	324
basicmath	174	173	pgp.enc	292	291
bitcount	73	73	qsort	94	91
blowfish.dec	55	56	rijndael.dec	107	104
blowfish.enc	55	56	rijndael.enc	107	104
crc	66	64	rsynth	232	236
dijkstra	87	85	sha	70	67
fft	124	120	stringsearch	80	80
fft.inv	125	131	susan.cor	91	89
ghostscript	971	971	susan.edg	103	100
gsm.dec	128	129	susan.smo	82	82
gsm.enc	176	175	tiff2bw	374	374
ispell	183	189	tiff2rgba	375	375
jpeg.dec	187	183	tiffdither	412	409
jpeg.enc	188	185	tiffmedian	368	368
lame	534	529	typeset	1052	1045

Fig. 6. Speedup of DBT/UCB relative to DE/FS

is disabled (bars "avg-demand"), the average 1.14 speedup of SPB is decreased to 1.12 (UCB-175%) and 1.11 (UCB-75%). From these results, the UCB has only a small loss relative to the SPB, yet it limits code space.

V. RELATED WORK

Embedded systems used to store code in NOR flash since it allows eXecute-in-Place (XiP). [15] showed that adding SRAM buffers to NAND flash can provide XiP. Otherwise, code in NAND flash must be loaded to main memory for execution. [2] shows that demand paging for NAND flash uses less memory and energy than full shadowing. [16] shows how to combine demand paging and XiP, using XiP only for infrequently executed pages. [17] reduces the time to handle a page fault by simultaneously searching for a page to replace and loading a new page into the NAND flash buffer. These approaches require an MMU to trigger a page fault.

The approach in [3] is the closest to ours since it also targets an MMU-less system. It uses a compiler to modify the binary before execution by changing calls/returns to invoke an application-specific page manager. DBT delays the code modification until run-time to handle binaries that have not been prepared in advance.

DBT infrastructures have been ported to embedded platforms and research has been done to make their code and data structure memory usage small [12], [5], [6]. [14] explored using scratchpad memory (SPM) and main memory for the F\$. The SPM can be managed as a software instruction cache. [7] use a binary rewriter to form cache blocks that are shadowed to main memory. A custom runtime copies code blocks to SPM during execution. [8], [5] achieve a similar effect by placing F\$ in SPM. [8] uses fragment compression and pinning to reduce the need to access flash for retranslation.

VI. CONCLUSION

This paper presented a novel software-only approach to demand paging for code stored in NAND flash, targeted to low-cost MMU-less embedded systems. The approach aims to overcome high start-up delay and poor amortization that can happen for full shadowing. In programs that do not amortize shadowing cost, our results showed at least a 50% reduction in boot-up time and an 1.11 average speedup.

REFERENCES

[1] A. Inoue and D. Wong, "NAND flash applications design guide," Toshiba America, March 2004.

[2] C. Park, J.-U. Kang, S.-Y. Park, and J.-S. Kim, "Energy-aware demand paging on NAND flash-based embedded storages," in *Int'l. Symp. on Low Power Electronics and Design*, 2004.

[3] C. Park, J. Lim, K. Kwon, J. Lee, and S. L. Min, "Compiler-assisted demand paging for embedded systems with flash memory," in *Int'l. Conf. on Embedded Software*, 2004.

[4] K. Hazelwood and A. Klauser, "A dynamic binary instrumentation engine for the arm architecture," in *Int'l. Conf. on Compilers, Architecture and Synthesis for Embedded Systems*, 2006.

[5] J. A. Baiocchi, B. R. Childers, J. W. Davidson, and J. D. Hiser, "Reducing pressure in bounded DBT code caches," in *Int'l. Conf. on Compilers, Architecture and Synthesis for Embedded Systems*, 2008.

[6] A. Guha, K. Hazelwood, and M. L. Soffa, "DBT path selection for holistic memory efficiency and performance," in *Int'l. Conf. on Virtual Execution Environments*, 2010.

[7] J. E. Miller and A. Agarwal, "Software-based instruction caching for embedded processors," in *Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2006.

[8] J. Baiocchi, B. Childers, J. Davidson, J. Hiser, and J. Misurda, "Fragment cache management for dynamic binary translators in embedded systems with scratchpad," in *Int'l. Conf. on Compilers, Architecture and Synthesis for Embedded Systems*, 2007.

[9] C. Park, W. Cheon, J. Kang, K. Roh, W. Cho, and J.-S. Kim, "A reconfigurable FTL (flash translation layer) architecture for NAND Flash-based applications," *ACM Trans. on Embedded Computer Systems*, vol. 7, no. 4, pp. 1–23, 2008.

[10] J. Chao, J. Y. Ahn, A. R. Klase, and D. Wong, "Cost savings with NAND shadowing reference design with Motorola MPC8260 and Toshiba CompactFlash," Toshiba America, July 2002.

[11] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. Davidson, and M. L. Soffa, "Retargetable and reconfigurable software dynamic translation," in *Int'l. Symp. on Code Generation and Optimization*, 2003.

[12] A. Guha, K. Hazelwood, and M. L. Soffa, "Reducing exit stub memory consumption in code caches," in *Int'l. Conf. on High-Performance Embedded Architectures and Compilers*, 2007.

[13] D. Ajwani, I. Malinger, U. Meyer, and S. Toledo, "Characterizing the performance of flash memory storage devices and its impact on algorithm design," in *Workshop on Experimental Algorithms*, 2008.

[14] J. A. Baiocchi and B. R. Childers, "Heterogeneous code cache: Using scratchpad and main memory in dynamic binary translators," in *Design Automation Conf.*, 2009.

[15] C. Park, J. Seo, D. Seo, S. Kim, and B. Kim, "Cost-efficient memory architecture design of NAND Flash memory embedded systems," in *Int'l. Conf. on Computer Design*, 2003.

[16] Y. Joo, Y. Choi, C. Park, S. W. Chung, E. Chung, and N. Chang, "Demand paging for OneNAND Flash eXecute-in-place," in *Int'l. Conf. on Hardware/Software Codesign and System Synthesis*, 2006.

[17] J. In, I. Shin, and H. Kim, "SWL: a search-while-load demand paging scheme with NAND flash memory," in *Conf. on Languages, Compilers, and Tools for Embedded systems*, 2007.