Dynamic Code Footprint Optimization for the IBM Cell Broadband Engine

Tobias Werth, Tobias Floßmann, Michael Klemm, Dominic Schell, Ulrich Weigand, and Michael Philippsen

1University of Erlangen-Nuremberg, Germany
2IBM Deutschland Research & Development GmbH
Programming Systems Group
Martensstr. 3, 91058 Erlangen
Computer Science Department
Schönaicher Str. 220, 71032 Böblingen

{werth, klemm, schell, philippsen}@cs.fau.de, t.flossmann@web.de, Ulrich.Weigand@de.ibm.com

Abstract

Multicore designers often add a small local memory close to each core to speed up access and to reduce off-chip I/O. But this approach puts a burden on the programmer, the compiler, and the runtime system, since this memory lacks hardware support (cache logic, MMU, ...) and hence needs to be managed in software to exploit its performance potential. The IBM Cell Broadband Engine (Cell B.E.) is extreme in this respect, since each of the parallel cores can only address code and data in its own local memory directly. Overlay techniques from the 70ies solve this problem with the well-known drawbacks: The programmer must manually divide the program into overlays and the largest overlay determines how much data the application can work with.

In our approach, programmers do no longer need to cut overlays. Instead, we automatically and at runtime fragment and load small code snippets into a code cache located in the local stores and supervised by a garbage collector. Since our loader does not load code that is not needed for execution, the code cache can be much smaller (up to 70%) than the original program size. Applications can therefore work on larger data sets, i.e., bigger problems. Our loader is highly efficient and slows down applications by less than 5% on average. It can load any native code without preprocessing or changes in the software tool chain.

1. Introduction

The Cell B.E. [6] is a heterogeneous multicore system. Eight Synergistic Processing Units (SPU) with a single-instruction multiple-data instruction set extend the single Power Processing Unit (PPU) that offers an extended PowerPC instruction set. Each SPU can access 256 KB of local memory (the local store) for both the executable code and the data. The Cell B.E. does not provide a shared memory programming model. Instead, the programmer must use explicit DMA requests to transfer data between the local stores and the memory sub-system of the machine.

Besides DMA transfers, the programmer also has to cope with the small local stores. As code and data reside in the same 256 KB area, it is crucial to keep the executable small. Smaller executables result in more space for data, fewer DMA transfers, and hence faster execution. Currently, the most effective way to create smaller code is to write and fine-tune it by hand. Even with techniques that semi-automatically generate overlays (see Sect. 2.1 for more details), for performance reasons the programmer wants to guide the code segmentation by hand.

In this paper, we propose an automatic solution that takes the task of finding an optimal overlay fragmentation away from the programmer. Our Dynamic Code Loader (DCL) defers decomposition until runtime by loading code into the local store on demand. While overlays have a statically defined size and will in general hold some code that is not needed for execution, DCL loads flexibly-sized code fragments into a code cache. Fragments may be as small as basic blocks—loading only actually required instructions—or as large as functions. In contrast to overlays, DCL avoids loading of seldom executed paths such as error handling routines. Garbage collection techniques automatically reclaim unused code fragments and free the code cache.

2. Related Work

Overlays, code caches, code compressions, and code size optimizations can help squeeze too large codes into a small memory. The latter two are orthogonal techniques that can be added to DCL to further shrink the footprint. The other two areas of related work will be discussed in more detail.
2.1. Overlays

Overlays and segmentation [13, 20, 23, 25, 28] are well-known techniques to execute binaries with a code footprint that is too large to fit into the machine’s main memory. During program execution, only a fraction of the code is kept in main memory. Currently unused code fragments reside on a slower secondary storage medium (e.g. a hard disk). If the execution path leads to an unloaded code fragment, an overlay manager retrieves the needed fragment from the secondary storage and replaces the currently loaded code fragment. Execution then continues until the next code fragment is needed for execution. In the following, we will give a short introduction on how overlays are implemented.

An overlay consists of (a set of) functions or may even contain only partial functions. Calls to functions within the loaded overlay can directly proceed. If, however, a function needs to be called that is defined outside the current overlay, it cannot be called directly because it is not loaded. Instead, a special proxy function is called that triggers the overlay manager. From the ID (e.g. an encoded function pointer) of a called function, the overlay manager deduces which overlay is needed, loads it, and resumes execution as described above. When returning from a function call, the caller’s overlay might have been replaced. In this case, the called function cannot simply return to the caller. The overlay manager first has to replace the callee’s overlay with the overlay the caller belongs to. Again, proxy functions are involved that trigger the overlay manager and inform it about which overlay to retrieve from secondary storage.

Although overlays provide an effective means for code footprint reduction, they have severe drawbacks. First, the largest overlay determines the memory footprint. As overlays are limited to (sets of) functions, overlays cannot be chosen arbitrarily small without worsening performance. Second, the best overlay decomposition can only be determined manually by the programmer. If chosen wrongly, the program suffers from a performance loss that is caused by overlay thrashing [8]. Third, since the executed code often shows a non-local behavior with respect to the instruction addresses, several overlays have to be replaced frequently. Fourth, overlaying wastes precious memory, since any fixed granularity will always contain code that is not used for the given input data. Finally, optimization is even more difficult for object-oriented code. Programmers tend to group methods according to their class memberships, which makes it hard to find a good decomposition scheme. Moreover, since in general it is impossible to predict the target of virtual function calls, construction of tight overlays is difficult.

Similar problems can be found in systems with a Memory Management Unit (MMU) [30]. Paging in and out program code can be seen as a transparent hardware implementation of overlays with less prominent problems. As MMUs provide transparent memory management, labor-intensive overlays lost their footing and almost vanished [9] until they have been revived for example for the Cell B.E.

DCL does not need a pre-processing step to organize code into overlays. It does not load unused code. Our garbage collector only removes small code fragments instead of overlays/pages that in general also contain code that should better stay in the memory to avoid thrashing.

DCL can also be used as an orthogonal extension of overlays. Instead of loading a fixed overlay, only the required code snippets from that overlay could be loaded by DCL. It is future work to combine the results of automatic code segmentation and dynamic loading.

2.2. Code Caches

We only discuss systems that do not rely on special hardware support to trigger loading or to perform translations between addresses in a given binary and in the code cache.

Due to the importance of scratchpad memory in embedded systems, quite a few techniques have been proposed there. Most of them use the scratchpad memory as a data cache only, for example [7, 11, 17]. In contrast to data caches, software caches can do better optimizations as there are fewer indirections and more knowledge about branch targets. Scratchpad systems that address code (e.g. [2, 21, 24, 29, 31]), in general apply profiling or static performance analysis to detect hot areas of the code. The compiler produces a statically optimized version of the hot code that the runtime system later loads into the scratchpad memory for execution. In the Cell B.E. area, all code must be in the code cache for execution, not just an optimized portion. Whereas omitted use of the scratchpad will only hurt performance on embedded systems, in our situation the code will not run at all. DCL does not rely on a static analysis for hot code and hence does not suffer from mispredictions. Finally, DCL works with any binary so that there is no need to change the tool chain.

Dynamic binary translators (e.g., [10]), dynamic code generators (e.g. [14, 22]), and run-time optimizers (e.g. [4, 5, 12, 27]) also use code caches to store the generated and/or optimized code and run it from there. Although we use many of the same ideas, the most important difference to our approach is that their code cache is large so that it can hold the results of trace optimizations, tail duplication, unrolling, inlining, etc. The code cache can be even 5 times larger than the original binary [15]. Hence, in contrast to DCL, the sizes of the loader and of all administrative data structures do not really matter. Neither is code garbage collection of similar importance, since their caches fill up seldom so that simple flush-when-full strategies are often sufficient. Another important difference is that these systems have a different goal. They spend significant time on opti-
mizations and transformations of the code so that the time spent for loading and for efficient management of the administrative data is of little importance. They will not easily throw away optimized code, since it is costly to reproduce. This is different in our domain. Our goal is to load and execute the given code with as little overhead as possible; reproduction cost of garbage collected code is small. Code unloading and efficient garbage collection is essential (see also [32]). Hence, it is the efficiency of the technical implementation that matters. In DCL, a basic trace optimization only happens as a cheap by-product of loading code snippets in the order they are used. There is only time for branch optimization that chains code snippets together.

The system that is closest to ours is the dynamic code loader presented by Miller and Agarwal [19]. Their approach is to pre-process the binary and separate it into basic blocks that are then split into fixed-size cache lines for loading; if the last cache line is not completely filled nops are appended. While Miller and Agarwal argue that it is crucial to move as much work as possible into a pre-processor and to add the runtime system at link time, we show that a fully dynamic approach can achieve (at least) the same runtime performance. The drawback of their approach is that they can only handle basic block granularity. However, our performance measurements show that a larger granularity is better. Furthermore, they either use a FIFO or a simple flush garbage collector, while we achieve better performance with a specific garbage collector for code. Since we do not add nop operations, our code cache is a bit smaller. That said, we have the same goals, use similar optimization techniques in the runtime system, but have an alternative general approach. On average, DCL has a runtime overhead of about 5%, while theirs is about 11%. Especially calls and indirect branches seem to be a bottleneck of their approach; the corresponding benchmarks show a runtime overhead of about 24%, whereas DCL has a worst case runtime overhead of 17%. Of course these measurements cannot be directly compared and can only be used as an indicator, but it seems that our dynamic approach causes a lower overhead. It is future research to find the perfect combination of both sets of ideas.

3. Dynamic Code Loader

Let us now describe the Dynamic Code Loader (DCL) in more detail. After an architectural overview, we discuss DCL’s main modules, before summarizing Cell B.E. specific aspects and limitations of our code loading approach.

3.1. Architectural Overview

The main design objective for our dynamic code loader is performance. Loading an SPU application with the loader should not cause severe performance drops during application execution. Otherwise, the benefit of Cell B.E.’s high performance SPU would deteriorate.

With this key requirement set, it becomes clear that a fully interpretive approach is not applicable to solve the problem of code loading. Instead, the SPU code has to run natively on the SPU core. The load on the PPU has to be kept as small as possible, as DCL must not slow down applications running on the PPU part. Even worse, for running applications with DCL on several SPUs, the PPU could become a bottleneck. But the pressure on the PPU is almost reduced to zero after a certain warm-up phase, when the application has started and most of the required code was loaded into the SPU’s code caches. Finally, DCL’s code footprint must be kept small, as it must be compensated by the savings of dynamic code loading.

Fig. 1 sketches the architecture of our loader. DCL is decomposed into three interacting modules: code analyzer, code cache, and garbage collector. The code analyzer accesses the host system and transfers code from the SPU executable on disk to the code cache. It also prepares the loaded code for execution from there. In addition to the loaded code, the code cache also maintains a directory of code fragments for administrative purposes. If the code cache runs out of code, special proxy functions trigger the code analyzer to load the next code fragment. Calls to those proxy functions are inserted by the analyzer in an earlier step. If the code cache is about to exceed its capacity, it triggers the Garbage Collector (GC) to make available new space (using a copying GC algorithm). The garbage collector accesses the cache, analyzes the loaded code fragments, and wipes out obsolete fragments that are currently not needed by the SPU application.

Fig. 2 shows the memory layout of a standard SPU program on the left. The program’s code starts at address 0 and is followed by the heap area in which memory allocated with malloc resides. The stack grows from the high ad-
addresses at 256 KB towards address 0. On the right, Fig. 2 shows the layout of a DCL-enabled SPU program. The code section is no longer present as the code is executed within DCL’s code cache. The loader’s code resides beyond the stack’s starting address, effectively pushing the stack towards address 0. The application’s heap now starts where the former code segment was.

The layout of Fig. 2 for DCL-enabled programs has two major advantages. First, DCL can allocate the code cache and all of its internal data structures during start-up by simply decrementing the stack pointer. Second, the implementation of malloc can keep the stack-heap collision check as is. If the code cache was allocated in the heap area, all implementations of malloc had to be made aware of a blocked memory range, effectively rendering existing C runtimes unusable with DCL.

3.2. Code Analyzer

The task of the code analyzer is twofold: (1) It reads and analyzes the loaded code to find boundaries of code fragments. (2) It prepares the loaded code for latter execution in the code cache. We will now discuss both tasks in detail.

In contrast to static overlay decomposition that needs a special compile-time pass, DCL decomposes the program into code fragments at runtime. It dynamically finds boundaries of the to-be-loaded code fragments by analyzing the SPU code while loading it. If loading is done on the level of basic blocks, the code analyzer stops loading a code sequence as soon as it reaches the first branch instruction (e.g., bra, brz). If the granularity is a function the analyzer seeks for $bi \extrm{ } 0$ instructions as they indicate a return.

The code analyzer uses the current allocation pointer (see Sect. 3.3) to find free space in the code cache and issues an fread call on the host system to transfer a chunk of code to the SPU’s local store. It then analyzes instruction

by instruction and compares them to a bit mask to detect an instruction that ends the fragment. If the last loaded instruction has been analyzed, the next chunk of instructions is transferred and analysis continues.

The analyzer also prepares the fragment for execution. As in overlay systems (see Sect. 2.1), branch instructions are replaced by proxy functions that trigger loading of the next code fragment. For each type of branch instruction, DCL provides a corresponding proxy function. Additionally, there is a “fall-through” proxy function to link to code that is reached without an explicit branch, i.e., by falling through a conditional branch. Fig. 3 shows a brz instruction (branch if zero) as an example. Path 1 represents the taken branch. If $s 12$ holds a non-zero value, execution falls through (path 2). The brz instruction is replaced by two proxy functions (Fig. 4). In case of path 1, the next code fragment is the basic block that begins at the instruction at target. The proxy loadNextBB is added to load the basic block that directly follows the brz instruction and that is executed if the branch is not taken. We will revisit proxy functions in the next section, as they are tightly coupled with the data structures of the code cache.

3.3. Code Cache

As already described, the code cache holds the prepared code for execution. It also stores an administrative data structure, the fragment directory, whose efficiency is crucial for executing the SPU program. The reason is that whenever the control flow reaches the end of a fragment for the first time, a proxy function is called that needs (two) directory lookups to determine the next fragment to be executed.

To locate fragments in the cache, the fragment directory keeps track of all loaded fragments. DCL uses the data

\footnote{Dynamically detected basic blocks may be larger than their static counterparts. In the dynamic case, the instruction-wise loading does not recognize labels and thus cannot know if there are branches into the middle of the currently analyzed block. DCL uses lookup tables that can handle branches that target instructions within such dynamic basic blocks.}
structure shown in Fig. 5 as directory entries. The `code` field contains a pointer to the fragment in the cache. The `file_offset` stores the fragment’s address in the original binary on disk. If the fragment is terminated by a branch instruction, the address of its target (in the binary) is kept in `target_file_offset`. The `size_and_flags` field holds the size of the fragment and flags (e.g., tags for the GC). The fragment directory is allocated at the end of the code cache and grows towards the begin of the cache.²

From the address of its invocation, the proxy function can find the entry of the currently executed fragment and the `target_file_offset` of the next block on the program’s execution path. This target address is also sought in the fragment directory. If it does not reside in the cache, the code analyzer loads it.

Obviously, it is crucial for performance that entries in the fragment directory are found as quickly as possible. We found hash tables to be inappropriate since (a) we need to search along different fields of the data structure, (b) we are dealing with multiple-entry, single-exit code regions, thus hashing over the fragments entry points provides only little benefit and (c) hash tables waste precious memory. Since it is more likely that the control flow exhibits some spacial locality instead of chaotic branches, it is reasonable that DCL achieves best performance with a linear array of struct fragment_t that is enhanced by a small cache that stores recently requested fragments. Only if a fragment was not found in this cache, the complete fragment directory has to be scanned. Measurements have shown that a capacity of only four elements suffices to make benchmarks with a large number of indirect branches four times faster. Additional 5% performance has been gained over a simple FIFO management by sorting the cache in a Least Recently Sought (LRS) fashion. Entries that have been sought recently reside at the beginning of the cache. As the flow of control may change and different fragments need to acquire the first place in the cache, the entries age if not sought for.

If DCL has determined the address of a loaded fragment in the cache or a fragment has been added to the code cache, a proxy function of a direct branch replaces its invocation with an appropriate branch instruction that directly targets the code fragment in the code cache. Hence, proxies of direct branches are only executed once, since afterwards the blocks are chained by means of direct branches.

Indirect branches (e.g., function pointer invocations) are a bit more complicated as they evaluate a register to determine the branch target. Hence, DCL cannot replace the proxy invocation with a direct branch to the code cache, as the target address might change for each execution of the branch. Since invoking the proxy function repeatedly would severely hamper performance, DCL uses an inline cache to quickly compare a target address against the address used in the last round of execution. If the target is unchanged, a direct branch to the fragment is taken. Only otherwise, the proxy function is called. In this case, the small cache mentioned above helps alleviate the lookup in the fragment directory, thus both techniques support each other.

3.4. Garbage Collector

We have experimented with basic strategies like FIFO and flush-when-full and found that a carefully designed garbage collector for code can achieve better performance. GC for code is different from GC for data. Whereas data objects that can be reached via references from the root set cannot be thrown away, reachable code fragments can. To do so, the proxy invocations must be re-patched back in (to undo the chaining). The code analyzer will automatically load a collected fragment again from the binary on demand. Only fragments that are referenced directly from return addresses on the stack or from registers may not be removed as there is no way to replace them with proxy invocations.³

Let us now discuss the different flavors of garbage collection [16] and their applicability to code collection. It is obvious, that reference counting does not work for code collection since the counters will never become zero as the application itself never frees any references to loaded code fragments. The weak generational hypothesis holds for code fragments as well, i.e., younger fragments die early [15]. However, on systems with small local memories several memory segments (hatchery, mature object space, etc.) are not affordable due to their memory requirements. From the different types of copying collectors, a mark&compact collector has turned out best [26].

Our mark & compact GC (MCGC) works well with small cache space and allows for efficient fragment allocation. First, MCGC traverses and tags all fragments that are still in use, i.e., that are reachable in the transitive closure of the root set via branch instructions. Afterwards, all

²Similar to heap and stack, the code cache area and the fragment directory dynamically grow towards each other and allocations can be easily performed by means of pointer arithmetics.

³Note, that to be conservative, every bit pattern on the stack that might be an address must be treated as a valid root reference. However, in the measurements we only saw very few false references.
unmarked fragments are reclaimed by moving the marked fragments to the beginning of the code cache, effectively deleting dead fragments. While moving fragments, the GC updates the entries in the fragment directory and re-writes the target addresses of branch instructions. As a result, new objects can be allocated at the end of the used space. An allocation simply shifts the pointer to the free memory.

Since the GC in general cannot correctly distinguish machine addresses from data values in the root set, it cannot rewrite them when moving the corresponding fragments. To solve this problem, DCL uses a trampoline for function calls. Similar to dynamically linked ELF shared libraries [18], a function call targets a trampoline entry which in turn invokes the called function. The GC can safely rewrite the trampoline entry without affecting addresses in the root set, as the trampoline’s entry address is fixed.

Only if a plain run of MCGC does not free enough memory, DCL removes some fragments that can be reached from roots deeply buried in the stack by re-patching the corresponding branches into proxy invocations. We can safely assume that this will not happen too soon, since the references are deeply buried in the stack.

3.5. Cell Specific Aspects

Callback Functions: An SPU can call(back) a function on the PPE, e.g. printf, open, etc. This is implemented in the library by means of the callback-helper _send_to_ppe that places code onto the stack and runs it from there. An identification of the function to be invoked by the PPE is also pushed onto the stack. The PPE then accesses the SPU’s program counter, reads the parameter from the SPU’s stack, and switches to the appropriate function.

DCL cannot use that approach, since to DCL the invocation of the helper on the stack appears to be a regular function invocation that triggers the code analyzer that then cannot resolve the library address in the application’s binary.

The solution is to bind the application with a new implementation of _send_to_ppe_new. This version calls the original implementation that is bound to the loader. The original version is therefore no longer treated by the loader’s address translation mechanism. To clearly separate the loader from this SPU library routine, we add an extra level of indirection, i.e., we add another table with function pointers (similar to the trampoline table).

Double Issue: An SPU uses an even and an odd pipeline for program execution. Every instruction can only be executed by one of the pipelines. If an even instruction is loaded from an odd memory address (or vice versa), there is a pipeline stall.

When DCL allocates a new block in the code cache, the loader automatically inserts nop instructions where needed to achieve the proper alignment of the loaded code.

Branch Hints: An SPU does not provide branch prediction hardware. Instead, it relies on explicit branch prediction instructions inserted by the compiler. Wrong, too late, or missing branch prediction hints cause pipeline stalls and a significant loss of performance.

It is therefore crucial for performance that DCL also rewrites branch predictions. Whenever a proxy function replaces its own invocation with a branch, the preceding branch prediction needs to be patched accordingly. Fragment moving during GC also requires re-targeting of the affected branch predictions.

3.6. Restrictions

DCL cannot handle arbitrary code. There are a few restrictions, some with a work-around.

Self-modifying Code: With the exception of the call-back functions (see Sect. 3.5), DCL does not support self-modifying code. The reason is that modified fragments cannot be reproduced as soon as the GC has removed them from the code cache. A possible solution is to place code in a special section that is marked as non-collectable within the SPU binary and thus easily detected by DCL. A future implementation of DCL could provide an API to enable this feature for an application.

Unmanaged Code: Unmanaged code is not located in the code cache and thus confuses the code analyzer’s address translation. Since this is similar to the _send_to_ppe problem, the additional trampoline will be used in future when there will be an API for programmers to inform DCL about the existence of such code.

Position Independent Code (PIC): Whereas a non-PIC program (i.e., regular code) uses absolute addresses, PIC relies on addresses that are relative wrt. to a given address, e.g. the base address of a library’s code. Traditionally, compilers emit code that stores the instruction pointer in a global variable (or a register) [1]. All subsequent branches use this variable as the base offset when calculating their targets. As DCL rearranges the code’s physical layout while a program is being executed, the relative offset is no longer correct. We feel that this restriction is acceptable as most SPU programs are linked against static libraries only.

Code Size Limitations: The current tool chain of the Cell B. E. SDK does not allow to build and to link executables larger than 265 KB. As DCL relies on the SDK’s standard linker, it is subject to this restriction as well. We will discuss a future combination with IBM’s or GCC’s overlay manager in Sect. 6.
4. Performance Evaluation

We have evaluated the prototype of DCL on four Fortran codes of the NAS Parallel Benchmark suite [3] using the IBM xlf compiler. The other codes in that suite are larger than 256KB and do not run on the Cell B. E. (neither with DCL nor without). Our test environment is a PlayStation 3 equipped with a Cell B. E. processor.

We compared the native execution times (without DCL) and the runtimes with DCL and code caches of various sizes. Moreover, we have studied the effects of the code block granularity.

A first look at the results in Fig. 6(a) reveals that each plot has two steep corners. If the code cache is large enough the benchmark is executed with a reasonable overhead, i.e., 4-5% on average (on the right hand side of the plots). This runtime overhead is negligible for three of four cases and still reasonable for the last one. The size of the code cache is about 20-30% of the size of the native executable. If the size of the cache is reduced even more (to the left of the right steep corner), the code cache gets so small that there is a function fragment that does no longer fit—even with GC. DCL can continue to operate on basic block granularity but the runtime effects of more calls to proxy functions and more patching overhead are significant. Nevertheless, one can reduce the cache space further (if necessary and if the runtime overhead is acceptable).

The left steep corner shows another obvious effect: far too small code caches (i.e., smaller than 10%-20% of the original code size) cause painful runtime overheads. Obviously, with a too small cache too many GC cycles are needed. This happens as soon as the main loop of each of the four benchmarks demands more code than the code cache can take and, hence, each iteration causes at least one GC cycle plus subsequent re-loading of the same blocks.

The bird’s eye view on these numbers is that DCL with function granularity can save about 70% of the application’s footprint without a severe slowdown. With basic block granularity, DCL can work with even smaller cache sizes, but this will noticeably slow down applications.

The purpose of Table 1 is to gauge the patching overhead. For all the benchmarks, we have measured the sizes of the loaded blocks both before and after patching. We did these measurements both for a loader that worked on basic block granularity and for one that operates on function granularity. It can be seen that the latter deals with code blocks that are about three times as big. Since on average, patching adds 4-8 bytes, i.e., 1-2 instructions to a block, it is a smaller burden for function granularity where only 1-2% of the total memory is used for patching. It fits well to the explanation of the rightmost steep corners in Fig. 6 that they are steeper when the fragment growth in going from basic block granularity to function granularity is more prominent. Compare for example the plots of MG (147.8 → 511.7, factor 3.5) and EP (130.0 → 259.3, factor 2). Larger fragments result in fewer calls to proxy functions and less overhead.

Furthermore, we have tried hard to manually partition the benchmarks into overlays. Results are given in Table 2. This is a difficult task and although we have identified some opportunities for overlays, the root segments of the resulting overlay structures have still been quite large.

DCL has always outperformed our manual endeavors and has automatically achieved much smaller code caches for the same benchmarks. Moreover, three of four benchmarks run faster with DCL (at least with function granularity) than with the overlay manager.

5DCL is not restricted to a specific programming language. It can load and analyze the assembly code, i.e., the full instruction set of the SPUs.

5. Conclusions

We have presented DCL, an automatic dynamic code loader for the Cell B. E. that is not only feasible and efficient but also significantly reduces the memory footprint of applications. DCL only loads code blocks that are actually
needed at runtime. The loaded code resides in a code cache that is allocated in the local store of an SPU. The loader uses function granularity for code loading but sometimes falls back to basic blocks if a function is too large to be loaded in one piece. A mark & compact garbage collector manages the free space in the code cache and automatically unloads currently unused fragments when the cache fills up. On benchmarks, our loader adds a runtime overhead of about 5% on average to the application. We consider this to be a small price to pay for code size reductions of about 70%. DCL is completely transparent to the application programmer and hence eliminates the difficulties caused by small local on-chip memories, featured by many of today’s embedded and some multicore processors.

6. Future Work

DCL can still be improved in some key areas.

DCL uses trampoline functions to redirect function calls (see Sect. 3.4). This approach makes our GC independent of a particular hardware platform and ABI. On the Cell B. E., return addresses live in fixed and known stack slots. We could save the trampoline and the corresponding housekeeping directory. However, it needs to be studied whether trampoline indirection or stack traversal is faster.

Currently, DCL only patches the branch hints that are already present in the code. However, since DCL knows about proxy invocations and the branches at the end of a code fragment, DCL could insert additional prediction hints. As the newly inserted hints must have a certain distance from the branch to get optimal results, some new hints must be added in the preceding fragment. It remains to be investigated whether the runtime costs of additional DCL code can be bought back by better branching performance.

DCL’s most important issue is the 256 KB limit for SPU program size. Since DCL uses the standard tool chain, it is not possible to create binaries exceeding 256 KB. Overlay managers circumvent this constraint by placing all overlays at the same address. Currently, we investigate how DCL can work with an overlay manager. We use overlay techniques to decompose the application into smaller overlay segments. Within such segments, DCL dynamically loads code to further decrease memory consumption. In contrast to using just overlays, not only less memory will be needed to execute the code but also thrashing effects may be avoided.

Inspired by the related work [19], we will investigate which parts of DCL can be moved to a pre-processor. Aside from computing the fragment’s boundaries prior to execution, another candidate is the detection of good places to insert additional branch prediction hints, as mentioned above.

Finally, there might be ways to improve the fragment directory lookup. We investigated hash maps that provided some gains but are prohibitively expensive wrt. memory consumption. For the decision whether a fragment resides in the code cache, DCL scans the directory in order to answer this question in the worst case. A future version of DCL should come up with a more efficient algorithm, effectively avoiding this worst case.

References


