

# Multiple Page Size Modeling and Optimization

Călin Cașcaval    Evelyn Duesterwald    Peter F. Sweeney    Robert W. Wisniewski  
IBM T.J. Watson Research Center

## Abstract

*With the growing awareness that individual hardware cores will not continue to produce the same level of performance improvement, there is a need to develop an integrated approach to performance optimization. In this paper we present a paradigm for Continuous Program Optimization (CPO), whereby automatic agents monitor and optimize application and system performance. The monitoring data is used to analyze and create models of application and system behavior. Using this analysis, we describe how CPO agents can improve the performance of both the application and the underlying system.*

*Using the CPO paradigm, we implemented cooperating page size optimization agents that automatically optimize large page usage. An offline agent uses vertically integrated performance data to produce a page size benefit analysis for different categories of data structures within an application. We show how an online CPO agent can use the results of the predictive analysis to automatically improve application performance. We validate that the predictions made by the CPO agent reflect the actual performance gains of up to 60% across a range of scientific applications including the SPEC-cpu2000 floating point benchmarks and two large high performance computing (HPC) applications.*

## 1. Introduction

In this paper, we present a paradigm for *Continuous Program Optimization* (CPO), whereby applications and their execution environment are optimized throughout their lifetime. A key component of the CPO paradigm is a vertical *Performance and Environment Monitoring* infrastructure [29] that collects integrated performance information from any layer of the execution stack. This performance information is used by CPO agents to create models of the application and system performance. An analysis based on the models is then used to affect both the application code and the execution environment to increase system performance. In this paper,

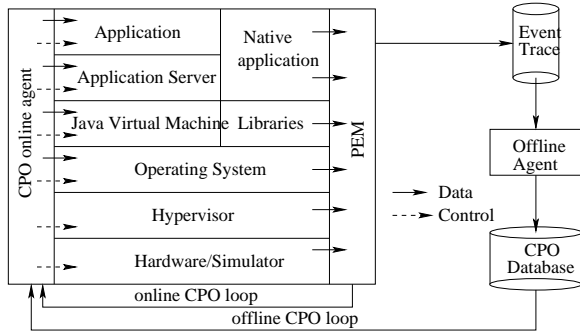
we focus on the benefits of CPO for single application performance in scientific and high performance computing. However, CPO is a general paradigm that targets a broad range of application scenarios including system throughput for commercial and transaction processing applications.

To demonstrate the CPO paradigm, this paper presents an offline CPO agent and an online CPO agent that cooperate to optimize large page usage. The first time an application is run on the system, the offline agent collects sampled data addresses and TLB (Translation Lookaside Buffer) misses using hardware counters, page fault information from the operating system, and memory allocation events from the runtime library. The offline agent uses this information to model the behavior of the application and predict the relative benefit of using large pages for different application data structures. The analysis results are stored in the CPO database by the offline agent. For subsequent invocations of the application, the online agent examines the database and the number of large pages currently available and determines which data structures to map to large pages. Consistent with the CPO paradigm, these subsequent runs are monitored by the online agent to validate that the chosen mapping had the predicted benefits.

We chose this page size optimization scenario because, although both modern hardware and most operating systems support multiple page sizes for improved performance, they are typically difficult to use, and therefore end users often do not take advantage of them. Our solution is completely automatic: no programmer intervention is required.

In this paper we make the following contributions:

- we present a validated multiple page size benefit analysis; we show how using information across layers of the execution stack allows an offline agent to predict the behavior when application data structures are mapped to different page sizes; we validate that the execution time of the applications follows the predicted benefits; and we show this validation holds across different inputs.



**Figure 1. Architecture of CPO and PEM**

- we describe an online agent that implements data allocation and page mapping policies based on the model predictions.
- we demonstrate the benefits of the CPO approach for page size optimization through experimentation.

We demonstrate our analysis on a combination of the SPECcpu2000 floating point benchmarks [22] and two large HPC applications (UMT2K [27] and RFCTH [13]). The infrastructure is implemented on the K42 research operating system [3, 15] and was run on Apple Xserve G5 systems [1].

In this paper, the page size optimization scenario is presented as an instance of the CPO paradigm. The paradigm is designed to support additional optimization scenarios. For example, we are exploring MPI communications as our next optimization scenario.

The rest of the paper is organized as follows: in Section 2 we describe the CPO paradigm; in Section 3 we present our analysis for estimating the benefits of multiple page sizes; in Section 4 we describe the online agent; in Section 5 we discuss our infrastructure and we present the analysis validation. We present related work in Section 6 and conclude in Section 7.

## 2. The CPO Paradigm

Continuous Program Optimization is a paradigm to assist in and automate the challenging task of performance tuning. In CPO, performance tuning is a continuous process of feedback-directed adaptation along two dimensions: 1) adapting the application to its execution environment, and 2) adapting the execution environment to enhance application performance. While the idea of feedback-directed optimization in compilers is not new, CPO extends it to a more dynamic, negotiation-based optimization process beyond compiler based optimization.

CPO optimizes programs throughout their lifetime and programs may switch between tuning and monitoring during execution or may be optimized between runs. The CPO paradigm involves the following: 1) dynamically identifying and characterizing performance problems; 2) modeling application behavior and negotiating resources on behalf of the application; 3) applying optimizations based on resource availability; and 4) validating that the applied optimizations are effective.

There are several components of the CPO architecture (shown in Figure 1) that execute in the two primary phases of tuning and monitoring. The PEM infrastructure [29] allows vertically integrated performance monitoring information to be gathered from all layers in the execution stack. CPO agents analyze this data and perform optimizations based on the analysis. The analysis consists of modeling salient aspects of application behavior, using static information about the application, and obtaining dynamic data from PEM. Some CPO agents execute online while others execute offline. Offline CPO agents are used to optimize applications between runs, and are usually invoked once, unless the optimization fails the validation phase. CPO agents, both offline and online, store behavior information in the CPO database.

The CPO database acts both as a repository for the history of past optimizations and their performance metrics, as well as a unified place to resolve competing or conflicting optimizations. There may be several CPO agents active concurrently, and their resource requirements may be conflicting. The common database and PEM infrastructure provides the means for resolving potential conflicts by providing each agent with a view of performance events across the entire system. CPO agents can coordinate their activities by monitoring the system for performance events issued by other agents.

The validation of optimizations is a vital aspect of the CPO paradigm. It is crucial because modeling will not be able to capture all the complex interactions in current or future systems. Therefore, it is critical to continuously monitor the optimized applications in order to validate that the modeled optimizations are having the intended effect and take action if they are not. Both optimization and validation occur automatically without programmer intervention.

In this paper we demonstrate our first instance of the CPO paradigm for optimizing the use of large pages. We have implemented the core PEM infrastructure, an initial prototype of the CPO database, and offline and online CPO agents that cooperate to automatically exploit the benefits of large pages. The page size optimization scenario instantiates most but not all aspects of the CPO paradigm. Currently, the CPO agents achieve the

optimization through an offline optimization loop, that is, across two runs of the applications. Future work includes online optimizations during a single run for different CPO scenarios.

### 3. Page Size Benefit Analysis

Most current hardware and operating systems support multiple page sizes. The benefits of using large page sizes include reduced TLB misses and reduced page faults. On some architectures (but not the Apple Xserve G5 we used in our experiments) large pages can improve hardware prefetching if the prefetching mechanism extends to large page scope. However, larger page sizes come at a cost: potential increase in fragmentation and an increase in memory footprint which may cause additional paging overhead in the operating system, especially in a multiprogrammed environment. Large pages also provide a contiguous data layout in real memory which may benefit cache behavior and prefetching. Given these constraints, the decision of what portions of the application data should be mapped to different page sizes is non-trivial. Moreover, current interfaces in commercial operating systems and hardware have placed the burden on the end user to make the decision of which data to map to large pages, thus further limiting large page usability.

In this section we describe an automatic way of mapping data structures in an application to different page sizes. We first introduce our notation, then describe our prediction model and its implementation. In Section 4 we discuss how the model can be used to implement different policies of large page usage, and then we present the results validating the model on real applications in Section 5.

#### 3.1. Notation

We introduce the following notation:

- $E$ : event stream — the collection of vertically integrated events from the PEM infrastructure including memory allocation and deallocation events, page faults, DTLBs misses, and sampled memory addresses; Each memory allocation event in  $E$  denotes a unique data object.
- $c_i$ : data category — used to group different data objects that are processed as a single entity by the model. The granularity of the categories is configurable and can be as fine as individual data objects.
- $M$ : page size mapping — the function that determines a page size  $M(c_i)$  for each data category  $c_i$ .

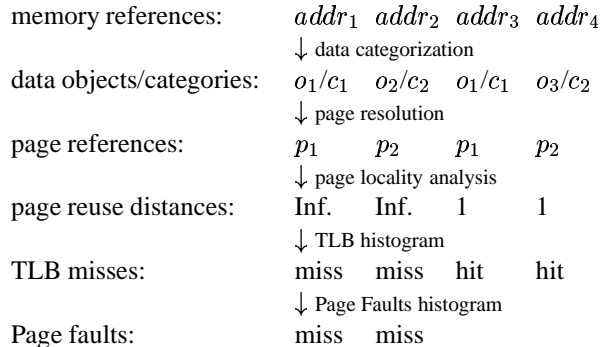


Figure 2. Processing of an event stream

For  $c$  categories and  $p$  different page sizes, there are  $p^c$  different possible mappings in an application. The model only considers a subset of the total mapping space: the *base mappings*, where a base mapping is a mapping that either maps all categories to the same page size, or a single category to a non-small page size.

- $RD_R(r_i)$ : the reuse distance of reference  $r_i$  in a stream of references  $R = r_1 r_2 r_3 \dots$ . The reuse distance is defined as the number of distinct references between two accesses to the same address [17]. For example, in the reference stream  $a b c c a$ , the reuse distance for the first access of  $a$  is *infinity* because  $a$  has not been accessed yet and the reuse distance for the second access of  $a$  is 2.
- $H_R$ : the reuse distance histogram for a reference stream  $R$ . Each entry  $H_R(i)$  in the histogram denotes the number of references with reuse distance  $i$ . Reuse histograms are a good metric of locality, and can be used to model cache and page behavior as follows: given a fully associative cache with an LRU replacement policy and  $C$  entries, the number of hits and misses for  $R$  can be computed as:

$$Hits_R = \sum_{i=0}^{i < C} H_R(i) \text{ and } Misses_R = \sum_{i \geq C} H_R(i). \quad (1)$$

#### 3.2. Analysis

Given a stream of memory references  $R \in E$  and a set of data allocation and deallocation events from the application, the page size benefit analysis estimates the reduction in the number of TLB misses and page faults for a set of page size mappings  $M_i$ .

The event processing algorithm is illustrated in Figure 2 and consists of the following steps:

**Data categorization:** for each reference  $r_j \in R$ , determine the data object  $o_i$  by mapping the address of  $r_j$  to one of the allocation events. In addition, determine the data category  $c_k$  for object  $o_i$ ;

The remaining steps are performed for each base page size mapping  $M_i$ . The model is additive, so that the benefits of non-base mappings that map more than a single category to a large page size, can be derived by combining the benefits of the respective base mappings.

**Page resolution:** translate reference stream  $R$  into a stream of page accesses  $P = p_1 p_2 p_3 \dots$  by modeling the mapping  $M_i$  for the given data categorization;

**Page locality analysis:** apply reuse distance analysis to the page accesses in stream  $P$ . Reuse distance computation is applied twice: once to compute the reuse distance histogram for TLB references,  $H_{TLB}$ , and a second time to filter the page accesses for those reference that miss in the TLB to compute the histogram for page faults,  $H_{PF}$ ;

The histograms obtained by processing the event stream are used in the following steps:

**Cost model computation:** determine the number of TLB misses and page faults and estimate the number of cycles serving misses as follows:

$$\begin{aligned} MissCycles(M_i) = & Misses_{H_{TLB}}(M_i) \times C_{TLBmiss} \\ & + Misses_{H_{PF}}(M_i) \times C_{PF}, \end{aligned} \quad (2)$$

where  $Misses_{H_{TLB}}(M_i)$  and  $Misses_{H_{PF}}(M_i)$  are computed using Equation 1 on  $H_{TLB}$  and  $H_{PF}$ , respectively<sup>1</sup>.  $C_{TLBmiss}$  and  $C_{PF}$  are the (experimentally determined) average numbers of cycles for a TLB miss and a page fault;

**Benefit ranking:** the last step of the algorithm, ranks the mappings  $M_i$  based on the number of pages of each size used and the  $MissCycles(M_i)$  value.

The benefit ranking computed by the algorithm is saved in the CPO database. The database entry consists of a set of directives, one for each data category explored by the analysis through the corresponding base mapping. Each directive denotes a data category and its computed benefit rank for different page sizes. The directives are used by the online CPO agent to implement different page mapping policies as outlined in Section 4.

<sup>1</sup>Equation 1 applies to fully associative TLB and page tables. Therefore, to estimate the misses for a TLB and page table we model a smaller sized fully associative TLB and page table, respectively, similar to [14].

### 3.3. Implementation

We have implemented the CPO agents for the page size benefit analysis and the PEM infrastructure on top of the K42 [15] research operating system running on Apple Xserve systems. These systems are dual processor (PowerPC 970FX [4]) machines.

The PowerPC 970FX processor supports two page sizes, 4 KB and 16 MB. K42 supports large page mappings as an extension to the mmap system call. K42 uses the GNU glibc [12] library. In this paper, we defined the data categories based on the glibc malloc allocation policy that distinguishes between small and large allocation requests. Thus we define three categories: static data (which includes BSS data), small dynamic allocations (allocations below 128 KB) and large dynamic allocations (allocations above 128 KB). Given the two page sizes and three data structures categories, we have five base page mappings to explore: (1) *all small*, (2) *static*, (3) *small dynamic*, (4) *large dynamic*, (5) *all large*. *All small* and *all large* map all data categories to small and large pages, respectively. *Static* maps only the static data category to large pages, and *small* and *large dynamic* map the small and large dynamic category to large pages, respectively.

**Event stream:** The PPC970FX processor has a set of programmable hardware performance monitors (HPMs). In addition to counting individual events, such as cycles, cache misses, and instructions completed, the PPC970FX supports the ability to sample instruction and data addresses for marked instructions. We use the sampling mechanism to collect data addresses of load instructions in the running application to generate a stream of memory references. In Section 5.1 we discuss how the sampling rate affects the accuracy of our analysis. During the same run we also collect data TLB miss information and we used K42 to collect information about the frequency of page faults. To produce dynamic data allocation and deallocation events we instrumented the *malloc* and *free* functions in glibc and we collected information about the static data segment of an application from its binary image.

**Significance test:** Prior to performing the page size benefit analysis we applied a *significance test* to rule out applications that have insignificant TLB miss and page fault rates and can therefore not directly benefit from large pages. An application fails the significance test if both its data TLB miss rate and its page faults rate are below certain thresholds (further discussed in Section 5.3.1). Passing the significance test provides a necessary but not a sufficient indication of benefits from large pages. Large page benefits depend on both a significant TLB miss/page fault rate and the application's page

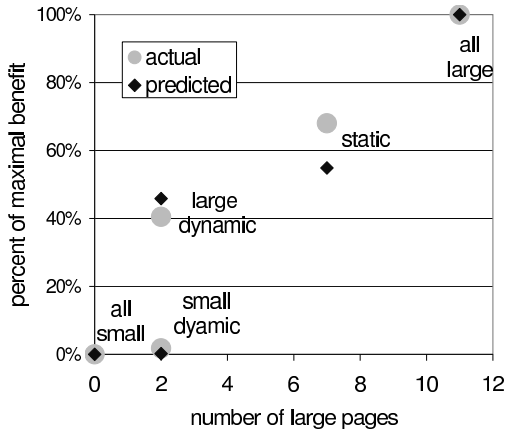


Figure 3. Benefit ranking for galgel

access pattern which is analyzed by our benefit analysis.

**Benefit ranking:** We perform the page size benefit analysis on the applications that pass the significance test. The outcome of the analysis is a benefit ranking. To provide detailed information for decision making in the CPO online agent, we define a new ranking strategy, a relative weighted benefit ranking, based on the *percentage of maximal benefits* (PMB) rank and the number of large pages that are required to fulfill that mapping. The mapping with the highest benefits compared to the default all small page mapping has a ranking of 100%. All other mappings will have a fraction of maximal benefits, between 0-100%. An example of the benefit ranking for the SPECfp2000 benchmark *galgel* is presented in Figure 3. The benefit ranking shows the percentage of maximal benefits against the number of large pages that are needed for each of the five base mappings.

In the special case that we run the application with the same input, the online agent can use the ranking to determine the most beneficial option for the number of currently available pages. In the more general case across different input, the online agent uses only the PMB ranking to select the most beneficial mapping. This way the agent optimizes large page utilization by avoiding to use large page for data categories with no or minimal predicted benefits.

#### 4. Page Allocation Policies

An important aspect of the page size benefit analysis is that it is predictive and not just reactive. After running an application once using the default configuration of the *all small* mapping, the offline CPO agent is able to predict the relative benefit for different data categories. The benefit ranking for each of the chosen category mappings is stored in the CPO database.

We implemented our online CPO agent prototype for a single workload scenario. When a particular application has been invoked, the online agent uses the stored benefit ranking to determine the best mapping of categories to large pages. Many operating systems currently limit the number of large pages available to applications. If there are insufficient large pages available to back all the application’s data, the online CPO agent selects only the categories of the application’s data that will benefit the most from large pages and directs the underlying operating system to implement that mapping. This is useful even in a dedicated environment; the automatic agent frees users from reasoning about whether their whole application can be backed by large pages.

**Discussion.** In a multiprogrammed environment, the online CPO agent faces a more complicated task, because unlike running in batch mode, the agent does not necessarily know a priori how many applications will be run or what their memory requirements will be. A possible heuristic for the online CPO agent in a multiprogrammed environment is to keep track of the number of available large pages and as the pool decreases, to increase the PMB rank that a data category must exhibit in order to be mapped to large pages. Being able to selectively choose on-the-fly data categories to map to large pages allows better use of large pages than if applications were written with a fixed mapping.

Currently, we work on architectures with two page sizes, 4 KB and 16 MB. Future systems will have a range of page sizes with intermediate page sizes such 64 KB, making a manual decision of what to back with which page size even more challenging. Thus, we expect agents that automatically select the most beneficial page sizes to be increasingly valuable in the future.

## 5 Experimental Results

### 5.1. Infrastructure

**Underlying operating system and tracing support:** We chose to develop our prototype on top of K42 [15] a research operating system. K42 is one of the operating systems recommended in the Department of Energy’s Operating/Runtime Systems for Extreme Scale Scientific Computation [3] initiative.

K42 provides an efficient foundation to support the Performance and Environment Monitoring (PEM) infrastructure. K42 has a mechanism allowing for events from anywhere in the execution stack to be logged to a unified trace buffer. The trace buffer is shared across all address spaces including the kernel with events written to it using a non-locking atomic update scheme [28]. The

implementation supports variable length events, which is important for space efficiency.

Another motivation for using K42, was that while it supports a Linux API allowing us to run standard benchmarks, it is also designed to be customized easily. Thus, the changes we needed to make for our large page policies were minimal. Now that we have completed our prototype implementation, we are implementing portions of the CPO framework on other operating systems.

**Large page support:** We implemented the support for the different page size mappings using environment variables to control each data category mapping. For the *static data* mapping we modified the K42 program loader to map the static data and BSS sections to large pages. For the *small dynamic category*, in which the space is allocated using the `brk` system call, we modified K42 to map the heap to large pages. For the *large dynamic* category, which glibc allocates through the `mmap` system call, modified glibc to pass an additional large page flag to `mmap`. Each category is independently controllable with a separate environment variable.

As in most commercial systems, the underlying PowerPC 970FX architecture does not support dynamic up- or downgrade between large and small pages. Thus, to dynamically change page size would require explicit copying. Due to the involved overheads, we only change the page size mapping across runs.

**Sampling rate and accuracy:** The CPO agent captures the application memory access patterns through sampled data addresses. The PowerPC 970FX architecture allows loads and stores to be marked, and upon completion records the address of the completed load or store. As with any statistical sampling there is a tradeoff between how frequently the addresses are sampled and the accuracy of the results. Furthermore, we wanted to verify that the sampled stream was representative of the full data reference pattern exhibited by applications.

We performed the following preliminary investigation. We ran the applications and the K42 operating system both on hardware and on a simulator. On the simulator we gathered every load and store address. On hardware we marked instructions to gather addresses with different granularities ranging from marking every fiftieth instruction to every thousandth instruction. Empirically we found this gathered about twenty times fewer loads and stores, i.e., marking every fiftieth instruction gathered about every thousandth address and marking every thousandth instruction gathered every twenty thousandth address.

After gathering the address streams from hardware and simulation, we ran a statistical analysis to determine the probability that the sampled address stream gathered on hardware is representative of the full address stream

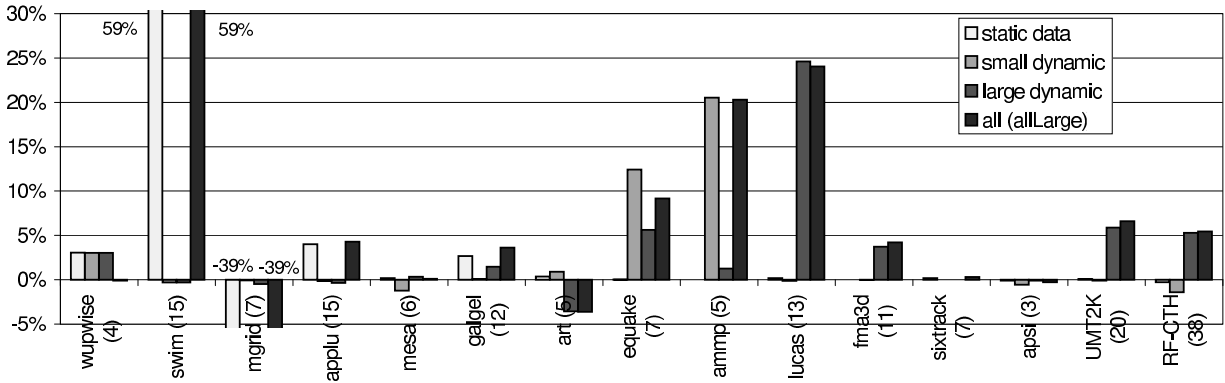
gathered on the simulator. The Hellinger Affinity Kernel (HAK) [2], loosely based on the Kullback-Liebler Divergence [21], takes as input two probability density functions (PDFs) and generates as output the probability that the PDFs are the same. Preliminary results indicate the validity of our approach. Specifically, we performed a HAK calculation for the RF-CTH benchmark by comparing two traces generated on hardware using two different sampling rates, one gathered by marking every fiftieth instruction and one gathered marking every thousandth instruction. This yields a HAK value of 60%. Comparing the hardware trace generated by marking every fiftieth instruction to the full simulator trace yields a HAK value of 54%. Comparing two non-similar traces, one from RF-CTH trace and one from a different benchmark (Radix) yielded a HAK value of 7%. Other HAK calculations demonstrated a similar trend providing confidence that the hardware sampled addresses are representative of the full access pattern. A full scientific study is out of scope of this paper and the subject of future work.

For our experiments we used a sampling rate of marking every thousandth instruction which experimentally provided the best balance between sampling accuracy and trace size.

**Overhead:** When the CPO agents are enabled, there are three potential sources of overhead: trace collection during the initial application run, trace analysis in the offline CPO agent, and implementation of the page size directives in the online CPO agent in subsequent runs. Our PEM implementation leverages K42's efficient tracing infrastructures [28] that supports event trace collection to memory with negligible overhead. The CPO agent can be invoked directly off of the memory trace buffers resulting in no measurable tracing overhead. We used the reuse distance algorithm by Almasi et al. [5] in our offline CPO agent which has a complexity of  $O(N \log M)$ , where  $N$  is the number of page samples and  $M$  is the number of unique page samples. The work of the online CPO agent consists of retrieving the the page size directives and setting the appropriate environment variable and which incurred no measurable overhead with our simple prototype database implementation. In a production system, the overhead of the online agent will be dominated by the database retrieval cost.

## 5.2. Benchmarks

We experimented with the SPECfp2000 [22] benchmark suite and two large scientific applications: UMT2K [27] and RF-CTH [13]. For each application we use a large (reference) and a small (train) input data set.



**Figure 4. Execution time comparing all small pages with different data categories mapped to large pages.**

UMT2K is a 3D, deterministic, multigroup, photon transport code for unstructured meshes. It solves the first-order form of the steady-state Boltzmann transport equation. The equation’s energy dependence is modeled using multiple photon energy groups, each using a collocation of discrete directions. The memory access pattern varies substantially for each direction, and the entire mesh is “swept” multiple times.

RF-CTH is a code used to explore the effects of strong shock waves on a variety of materials using many different models. The code simulates shock hydrodynamics equations. It can be run in two modes, one using adaptive mesh refinement, and one with a fixed mesh. The results presented in the next section are obtained using the latter mode.

The UMT2K and RF-CTH benchmarks are written in a combination of C and Fortran and use OpenMP and MPI. We ran the uniprocessor version of the code.

Although we experimented with the SPECint2000 benchmark suite, the data footprints of these benchmarks are too small to show any benefits from large pages. While a large data footprint is necessary to benefit from large pages, it alone is not sufficient as shown in the next section.

### 5.3. Evaluation

Figure 4 illustrates the change in execution time that result from mapping the different data categories to large pages over the default *all small* mapping. Each group of four bars represents a benchmark. The first bar shows the change in execution time for the *static data* mapping. The second and third bars represent the change in execution time for the *small dynamic* and *large dynamic* mappings, respectively. The last bar shows the change in

execution time for the *all large* mapping. The execution times were determined as the median time of five executions. The number shown in parenthesis next to each benchmark name is the number of large pages required for the *all large* mapping.

The data illustrates that a simple one-size-fits-all strategy for mapping large pages is not effective. First, the data shows that even within the class of scientific applications, improvements from large pages vary widely. Second, data footprint size does not provide a sufficient indicator for large page benefits. For example, swim yields the greatest speedup from large pages (59%) with a data footprint size of 15 large pages. There are other benchmarks with equal or larger footprints (applu, UMT2K, RF-CTH) that have significantly lower speedups (4%-6.6%). Most importantly, the data shows that no specific category always performs the best when mapped to large pages. For example, static data is the best category for swim, applu, and galgel. However, for equake and ammp, small dynamic is the best category. Finally, for lucas, fma3d, RF-CTH, and UMT2K, the greatest improvements are achieved with the large dynamic category.

Another important observation from the data is that using large pages may also have an adverse impact on performance resulting in a slowdown. Adverse effects may result from the changed memory layout, additional conflicts in the memory hierarchy and impact on speculation. Mapping data to large pages may produce a different data alignment which can both introduce and eliminate cache conflicts. Furthermore, the contiguous real memory layout that results from large pages may cause an increase in memory bank contention. Mgrid is the extreme example. There is a 39% slowdown when mgrid’s static data is mapped to large pages. Although, we measured increases in speculation activity and memory traf-

benchmarks	reference input mapping: all small (all large)			training input mapping: all small		
	execution time	PGFLT rate	DTLB(K) rate	execution time	PGFLT rate	DTLB(K) rate
RF-CTH	197.1 (186.4)	796 (66)	935 (0.8)	23.6	1,205	67
UMT2K	78.6 (73.6)	4,073 (40)	291 (3.9)	9.0	6,734	200
apsi	438.8 (439.9)	129 (126)	353 (348)	7.6	7,342	39
sixtrack	163.7 (163.1)	81 (40)	7 (1.5)	42.1	369	11
fma3d	296.9 (284.4)	126 (34)	181 (5.7)	161.9	32	0
lucas	13.8 (10.5)	3,912 (364)	1,646 (2.7)	1.0	4,097	20
ammp	737.8 (588.0)	30 (9)	1,080 (157)	100.4	115	1,093
equake	154.9 (140.7)	129 (50)	312 (0.7)	28.1	243	233
art	225.6 (233.7)	18 (14)	4 (0.2)	32.3	117	2
galgel	237.7 (229.1)	919 (12)	208 (0.2)	13.4	2,817	40
mesa	190.3 (190.1)	45 (33)	35 (0.8)	113.7	70	28
applu	257.9 (246.8)	201 (16)	252 (0.3)	11.9	764	357
mgrid	215.7 (301.3)	77 (8)	307 (0.6)	10.6	401	194
swim	1,516.6 (617.8)	34 (5)	2,877 (0.2)	30.7	615	5,311
wupwise	166.9 (167.1)	304 (303)	179 (179)	25.8	1,991	172

**Table 1. The execution time, page fault rate per second, and data TLB miss rate per second when all small (all large) pages are used. The data TLB misses are in thousands.**

fic, our investigation with the available PPC970FX hardware performance monitors did not provide a clear explanation for mgrid’s slowdown. For other benchmarks, the impact from adverse effects is less pronounced. For example, art has a 3.6% slowdown when large dynamic objects are mapped to large pages. Equake is another example of adverse effects where the speedup of mapping all categories to large pages is lower (9.2%) than the speedup of individual category mapping (12.4% for small dynamic).

A model inherently does not capture all possible effects that may be exhibited during actual application execution. Moreover, the presence of adverse effects is highly dependent on the specific architecture. Dealing with architectural idiosyncrasies and a model’s inherent incompleteness motivated the validation component in the CPO paradigm. Validation is implemented through the ongoing monitoring of key performance measures, such as IPC, TLB and cache miss rates, in order to verify that the predicted behavior matches the observed behavior of the running program. If a mismatch is found, the online CPO agent updates the CPO database entry to de-emphasize the recommendations made about this application for future runs. The online CPO agent may also start another agent that will perform a more in-depth analysis.

### 5.3.1 Significance Test

Recall, that prior to applying the page size benefit analysis, a significance test is applied to rule out applications that have insignificant page fault or TLB miss rates. Table 1 shows the information that was collected to apply the significance test. The column labeled **PGFLT rate (DTLB rate)** shows the page fault (DTLB miss) rate per second. We apply the significance test to filter out benchmarks that have both a particularly low DTLB miss and page faults rates. We experimentally determined the significance thresholds to be 100 K for the DTLB miss rate and 1000 for the page fault rate. For the reference input the following benchmarks do not pass the significance test: mesa, art and sixtrack. For the train input the mesa, art, sixtrack, and fma3d failed the test. After failing the significance test, we can safely determine that no measurable benefits can result from page faults and TLB miss reductions alone so that no further analysis is necessary.

### 5.3.2 PMB Ranking Error

The page benefit analysis is applied to the benchmarks that passed the significance test. The analysis produces a predicted PMB ranking. We also ran the benchmarks with the reference input for different page size mappings to compute an actual PMB ranking based on execution

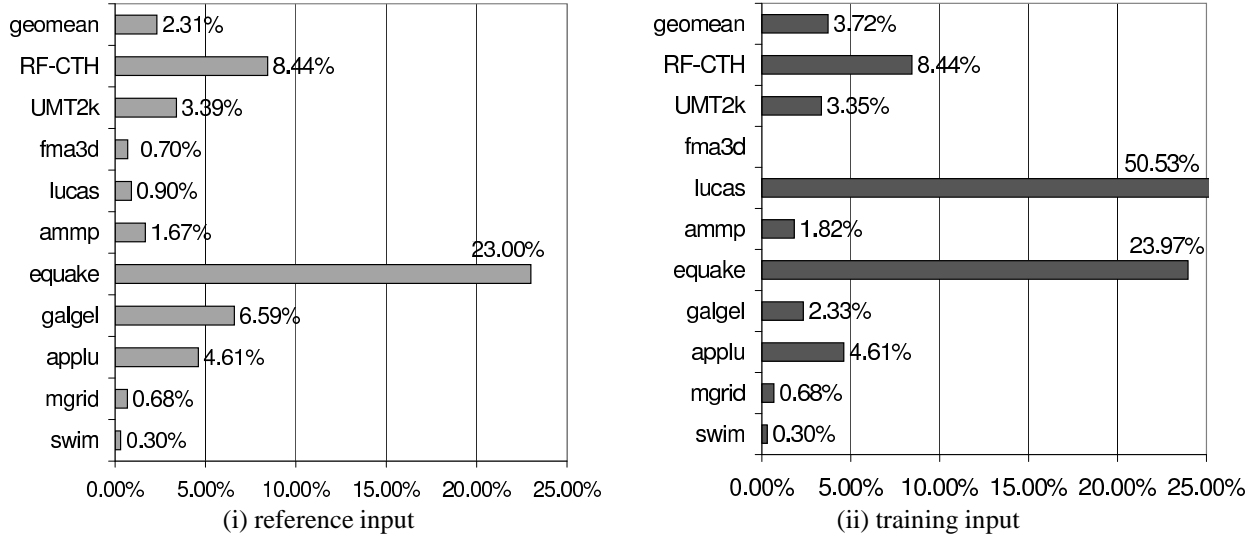


Figure 5. Benefit model prediction error for reference input (i) and training input (ii).

times as shown in Figure 4. We compute the ranking error per mapping to be the distance between the predicted and the actual PMB rankings. The overall ranking error for a benchmark is determined as the average ranking error across all mappings. A detailed view of the ranking comparison was shown for galgel in Figure 3, which results in an overall ranking error of 6.6%. The overall ranking error for all benchmarks is shown in Figure 5(i). The ranking error is low in most cases with a geometric mean of 2.3%. Equake is an outlier with a ranking error of 23%. As discussed earlier, equake is an example of a program where adverse effects dominate the performance improvements that result from reduced TLB miss and page fault rates.

There are two benchmarks (apsi and wupwise) for which no PMB rankings are produced because the analysis indicates no benefits based on the cost/benefit model. Figure 4 illustrates that there is no significant performance benefit from large pages for these applications. Inspection of hardware performance monitors in Table 1 for wupwise and apsi confirm that there is no significant change in TLB miss and page fault rates, validating the analysis prediction.

### 5.3.3 Input Sensitivity

There is evidence in the literature that locality properties of programs remain stable across inputs [9]. We conducted a second experiment that demonstrates that the computed PMB ranking that are based on locality properties also remain relatively stable across inputs.

We repeated the above experiment using the train input to produce the event trace for the analysis and compared

the resulting PMB ranking against the actual ranking for the reference input. Figure 5(ii) shows the resulting overall ranking error. As expected, there is a slight degradation in the geometric mean of the ranking error. However, in most cases the ranking errors are almost identical to the ones that results from same-input analysis.

There is one outlier, lucas, where we see a dramatic increase in ranking error. A closer inspection of lucas showed that in the train input the sizes of dynamic data structures are much smaller than in the reference input. In fact, in the training input, the large dynamic category was empty making it impossible for the analysis to match the benefits that the reference input obtains from that category. The lucas error points to a weakness in our current static scheme for data categorization based on data object size. We are currently working on a dynamic categorization strategy that uses access frequency and affinity information to group data structure into categories to address the problem with lucas.

## 6. Related Work

There is a significant amount of work related to feedback directed and dynamic optimization. A comprehensive survey of dynamic JIT compiler optimizations in virtual machines can be found in [6]. In the context of virtual machines, the term continuous compilation has been used to describe an approach in which compilation is overlapped with program interpretation and native execution [19]. Dynamic optimization has also applied to native binaries without compiler support [10].

There are a number of previous approaches that, like CPO, emphasize the aspect of continuity in the optimization process. The work by Kistler and Franz [16], describes an approach to continuous program optimization that has been applied to optimizing the data layout in object-oriented programs based on online feedback. Childers et al. [8] describe an approach to continuous compilation that applies aggressive code optimization at all times, from static optimization to online dynamic optimization. The continuous compilation framework has been demonstrated for adaptive application code in embedded systems. These previous approaches have focused on compiler optimizations. Our CPO paradigm goes beyond compiler optimizations and includes system and environment adaptation, such as the page size optimization described in this paper. Continuous optimization has also been used to describe a hardware dynamic optimization mechanism to optimize an application's instruction stream [11].

In the early 1990s researchers began analyzing the effect large pages had on the working set size and on TLB performance [25]. In later work researchers [24, 20] described the potential usefulness provided by large pages and identified hurdles in realizing the full potential. Romer et al. [20] states that "good policies for superpages have been elusive [because] a cost benefit analysis is required to determine if the overhead of constructing a superpage is outweighed by its benefit". Their work as well as more recent work [18] migrate data to large pages reactively with analysis performed at the operating system level using data gathered by the system. In contrast, we develop a predictive model and use information gathered from layers across the execution stack. Current operating systems such as Linux, AIX, and Solaris support restricted use of large pages. Commonly, a fixed pool of large pages is fixed at boot-time and may or may not be modified during system execution. If modification is allowed it is typically an expensive operation. Thus, the ability to predict the benefit of using large pages and the required number of such pages is helpful.

Stack distance or reuse distance has been used to model memory behavior since 1970 [17]. Following the initial implementation of stack distances using a list [17], researchers have proposed different methods to both speedup the stack processing and scale it to larger data sets [26, 7, 23, 5, 9]. Each method has its own trade-offs, based mainly on how the method is used: to model pages, TLB or caches. While the method in Ding and Zhong [9] is the fastest, it is also approximate. Because we are looking at pages and not cache lines, and the amount of data is not a concern, we chose to use the method in Almasi et al. [5] to compute the reuse distances for our page traces.

## 7. Conclusions

We presented a Continuous Program Optimization (CPO) paradigm leveraging vertically integrated performance data gathered from layers across the execution stack. As the complexity of hardware and software grows, and the speedup gained from hardware slows, there is a growing importance to work across the hardware-software interface, and across levels of the software stack. The CPO vision describes a paradigm whereby gathering data and affecting control are integrated across the stack.

A primary contribution of this paper was to use the CPO paradigm to implement offline and online CPO agents that cooperate using large pages automatically to improve application performance. The offline agent performs a page size benefit analysis and stores the analysis results in a database. The online agent uses those stored results to determine which data categories to request the underlying system to map to large pages. The process occurs automatically without manual programmer effort. We validate that the predictions made by the CPO agent reflect the actual performance gains of up to 60% across a range of scientific applications.

## 8. Future Directions

Our initial implementation divided data structures into fairly coarse categories. It may be advantageous to perform the page size benefit analysis on a per data structure basis. Furthermore, once the offline analysis is performed, the ability to integrate the analysis information with the compiler may allow a more efficient implementation of directing the mapping of data structures to large pages. We also would like to investigate a tighter integration with the operating system allowing it to call the CPO agent when it makes large page allocation decisions. The CPO agent may have a better opportunity to dictate policy than the operating system because the CPO agent has information gathered both from previous runs and across the whole execution stack. Finally, large pages is just one area where the CPO paradigm can improve performance. We are in the process of looking at other areas to apply the CPO paradigm.

## 9. Acknowledgments

This work was supported by Defense Advanced Research Project Agency Contract NBCH30390004. There have been many people who have contributed to the realization of the CPO vision: the entire K42 team, and in particular Maria Butrico, Michal Ostrowski and Bryan

Rosenburg, supported us throughout the implementation of the large page scenario with OS features and machines. Reza Azimi implemented hardware counter support for the PowerPC970FX in K42. Marina Biberstein helped with data visualization.

## References

- [1] Apple computer Xserve. <http://www.apple.com/xserve>.
- [2] Data clustering with kernel methods derived from Kullback-Leibler divergence. <http://www.sc.doe.gov/grants/Fr04-13.html>.
- [3] Operating/runtime systems for extreme scale scientific computation, department of energy. <http://www.sc.doe.gov/grants/Fr04-13.html>.
- [4] *IBM PowerPC 970FX RISC Microprocessor User's Manual*, 1.41 edition, November 2004.
- [5] G. Almasi, C. Caşcaval, and D. A. Padua. Calculating stack distances efficiently. In *Workshop on Memory System Performance (MSP 2002)*, Berlin, Germany, June 2002.
- [6] M. Arnold, S. Fink, D. Grove, M. Hind, and P. Sweeney. A survey of adaptive optimization in virtual machines. *Proceedings of the IEEE, Special Issue on Program Generation, Optimization and Platform Adaptation*, 93(2):32–56, February 2005.
- [7] B. T. Bennett and V. J. Kruskal. LRU stack processing. *IBM Journal for Research and Development*, pages 353–357, July 1975.
- [8] B. R. Childers, J. W. Davidson, and M. L. Soffa. Continuous compilation: A new approach to aggressive and adaptive code transformation. In *NFS Workshop on Next Generation, in Conjunction with IPDPS'03*, Nice, France, April 2003.
- [9] C. Ding and Y. Zhong. Predicting whole-program locality through reuse distance analysis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'2003*, pages 245–257, San Diego, CA, June 2003.
- [10] E. Duesterwald. Design and engineering of a dynamic binary optimizer. *Proceedings of the IEEE, Special Issue on Program Generation, Optimization and Platform Adaptation*, 93(2):32–56, 2005.
- [11] B. Fahs, T. Rafacz, S. J. Patel, and S. S. Lumetta. Continuous optimization. In *32nd Annual International Symposium on Computer Architecture (ISCA)*, Madison, WI, June 2005.
- [12] The GNU C library. <http://www.gnu.org/software/libc>.
- [13] E. Hertel, J. Bell, M. Elrick, A. Farnsworth, G. Kerley, J. McGlaun, S. Petney, S. Silling, P. Taylor, and L. Yarrington. CTH: A software family for multi-dimensional shock physics analysis, 1993.
- [14] M. D. Hill and A. J. Smith. Evaluating associativity in CPU caches. *IEEE Transactions on Computers*, 38(12):1612–1630, December 1989.
- [15] The K42 operating system, <http://www.research.ibm.com/k42/>.
- [16] T. Kistler and M. Franz. Continuous program optimization: A case study. *ACM Transaction on Programming Languages and Systems (TOPLAS)*, 25(4):500–548, July 2003.
- [17] R. L. Mattson, J. Gecsei, D. Slutz, and I. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2), 1970.
- [18] J. Navarro, S. Iyer, P. Druschel, and A. Cox. Practical, transparent operating system support for superpages. *SIGOPS Oper. Syst. Rev.*, 36(SI):89–104, 2002.
- [19] M. P. Plezbert and R. K. Cytron. Does just in time = better late than never. In *Proceedings of 24th Annual Symposium on Principles of Programming Languages (POPL)*, January 1997.
- [20] T. H. Romer, W. H. Ohlrich, A. R. Karlin, and B. N. Bershad. Reducing TLB and memory overhead using online superpage promotion. In *ISCA 1995 - International Symposium on Computer Architecture*, pages 176–187, 1995.
- [21] K. S. *Information Theory and Statistics*. Dover Publications Mineola New York, 1959.
- [22] SPECcpu2000 benchmark suite. <http://www.spec.org/cpu2000>.
- [23] R. A. Sugumar and S. G. Abraham. Set-associative cache simulation using generalized binomial trees. *ACM Trans. Comp. Sys.*, 13(1), 1995.
- [24] M. Talluri and M. D. Hill. Surpassing the TLB performance of superpages with less operating system support. In *ASPLOS 1994 - Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 171–182, 1994.
- [25] M. Talluri, S. I. Kong, M. D. Hill, and D. A. Patterson. Tradeoffs in supporting two page sizes. In *ISCA - International Symposium on Computer Architecture*, pages 415–424, 1992.
- [26] I. L. Traiger and D. R. Slutz. One-pass techniques for the evaluation of memory hierarchies. Technical report, IBM T. J. Watson Research Center, 1971.
- [27] The UMT benchmark code. <http://www.llnl.gov/ascii/purple/benchmarks/limited/umt>.
- [28] R. W. Wisniewski and B. Rosenburg. Efficient, unified, and scalable performance monitoring for multiprocessor operating systems. In *Supercomputing*, Phoenix Arizona, November 17-21 2003.
- [29] R. W. Wisniewski, P. F. Sweeney, K. Sudeep, M. Hauswirth, E. Duesterwald, C. Caşcaval, and R. Azimi. Performance and environment monitoring for whole-system characterization and optimization. In *PAC2 - Power Performance equals Architecture x Circuits x Compilers*, pages 15–24, Yorktown Heights, NY, October 6-8 2004.