Science of Computer Programming 98 (2015) 551–588

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Reducing operational costs through MIPS management

Łukasz M. Kwiatkowski[*], Chris Verhoef

Department of Computer Science, Vrije Universiteit Amsterdam, De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands

HIGHLIGHTS

- We present an approach to reducing CPU usage costs in large organizations.
- Source code scans are used to identify low-risk high-yield SQL-tuning opportunities.
- Our approach is suitable for deployment in large business-critical IT-portfolios.
- Case study: a mainframe production environment of 246 applications (19.7M+ LOC).

ARTICLE INFO

Article history:
Received 12 April 2012
Received in revised form 27 August 2014
Accepted 1 September 2014
Available online 10 September 2014

Keywords:
IT-portfolio management
Cost reduction
MIPS
Source code analysis
SQL

ABSTRACT

We focus on an approach to reducing the costs of running applications. MIPS, which is a traditional acronym for millions of instructions per second, have evolved to become a measurement of processing power and CPU resource consumption. The need for controlling MIPS attributed costs is indispensable given their significant contribution to operational costs. In this paper we investigate a large mainframe production environment running 246 Cobol applications of an organization operating in the financial sector. We found that the vast majority of the top CPU intensive operations in the production environment involve the use of DB2. We propose approaching portfolio-wide efforts to reduce CPU resource consumption from the source code perspective. Our technique is low-risk, low-cost and involves SQL code improvements of small scale. We show how to analyze a mainframe environment in an industrial setting, and to locate the most promising source code segments for optimizing runtime usage. Our approach relies on the mainframe usage data, facts extracted from source code, and is supported by a real-world SQL tuning project. After applying our technique to a portfolio of Cobol applications running on the mainframe our estimates suggested a possible drop in the attributed monthly CPU usage by as much as 16.8%. The approach we present is suited for facilitation within a mainframe environment of a large organization.

1. Introduction

Despite the often cited essay in the Harvard Business Review stating that IT doesn't matter [9], information technology plays an important role in many organizations. In our contemporary world, business environments have become global and there is a major challenge to deliver adequate computing services which meet stringent performance goals and operate at low cost. It is a wrong assumption that IT operations have already been optimized to the point that there is nothing to gain by further improving them as this paper will illustrate. There are still more than enough opportunities to achieve significant savings. A quote by Benjamin Franklin – "A penny saved is a penny earned" – applies very well to current business reality.

[*] Corresponding author.
E-mail address: lukasz.x@cs.vu.nl (Ł.M. Kwiatkowski).

http://dx.doi.org/10.1016/j.scico.2014.09.001
0167-6423/© 2014 Elsevier B.V. All rights reserved.

The more is saved on IT-operations, the more capital there is available to invest in pursuing new business opportunities. IT related operational costs are high and offer ample possibilities for cost reduction. For the Dutch banks it was estimated that total operational IT costs oscillate at around 20%–22% of total operational costs [6]. Businesses have taken a strong stance on operational cost reduction and seek solutions to slash IT costs. For instance, Citigroup estimated that the removal of redundant systems from their IT portfolio would yield a savings potential of over 1 billion USD [25]. So far, organizations have tried a number of approaches to cut IT costs which include staff reduction, outsourcing, consolidating data centers or replacing old software and hardware with its newer counterparties. However, all these approaches carry an element of risk and might end-up costing more especially when complex software environments are involved. Migrating IT-systems to another environment is expensive and full of risks. Computer software is highly dependent on the environment it is running in and changing that environment means identifying and removing those dependencies. On the other hand, there exists an approach which allows for IT costs to be reduced at low-risk for both business and IT. It involves management of CPU resource consumption on the hardware platforms and is the subject of this research contribution.

The MIPS metric is a traditional acronym for millions of instructions per second. It has evolved to become a measurement of processing power and CPU consumption. MIPS are typically associated with running critical enterprise applications on a class of computers known as mainframes. The term originates from the compartments in which these computers used to be housed: room-sized metal boxes or frames [19]. From a business perspective mainframes have proven to be secure, fast and reliable processing platforms. While running computer software costs money on every computing platform for mainframe users the incurred costs are relatively high. As Nancy White, a former CIO of Certegy corporation, once said: "MIPS and salaries are my highest unit cost per month" [11]. Mainframe users have traditionally chosen to pay usage fees because of the shared nature of the platform (many users use one big computer). However even when they run their programs in a large cluster of virtual machines they are charged on the capacity they reserve and/or use. Moving to another platform (from mainframes) might bring cost savings, but such a step is not risk free.

The amount of MIPS used by the average IT organization is on the rise. IT industry analysts estimate that most large organizations utilizing mainframes should expect their systems' CPU resource consumption to increase by 15–20 per cent annually. A Macro 4 project manager, Chris Limberger, explains financial consequences of this increase as follows [1]:

> Each additional MIPS typically costs around GBP 2500 in hardware and software charges. So if a company running a 10,000 MIPS system increases capacity by as little as ten per cent per annum, the incremental cost will be in the region of GBP 2.5 million. That's pretty typical but if your business is growing and if you're upping the level of activity on your mainframe, you can expect much more.

Despite the fact that the incurred usage costs are substantial, monitoring of CPU usage is not routinely implemented. So sometimes customers have no idea where the CPU resources are being consumed. The large availability of hardware and software tools for CPU cycle usage monitoring shows strong demand. Still it turns out that majority of managers (58%) admit that they do not continually monitor consumption of CPU resources [10]. Given the substantial cost implications for a business, the need for introducing measures which lead to reduction of MIPS utilization is indispensable.

1.1. Targeting source code

Where does the accrual of the amount of MIPS used happen? MIPS usage is directly driven by CPU usage, and CPU usage depends primarily on the applications' code. Inefficient code of the applications is considered to be a major cause of MIPS usage [10]. Therefore by improving performance of the code it is possible to significantly lower CPU resource consumption. For instance, software run on mainframes typically constitutes management information systems. For this class of software interaction with a database is ubiquitous. At the source code level interaction with relational database engines is typically implemented using Structured Query Language (SQL) which was specifically developed for interfacing with relational systems [43]. As a general rule of thumb most (≈80%) performance hampering problems are traceable to the SQL code [11].

Efforts aimed at optimizing the software assets' source code are a viable option for limiting CPU resource usage. In fact, code improvement projects have yielded operational cost savings. According to [12] a financial services company identified two lines of code that, once changed, saved 160 MIPS. Given the market price of MIPS the two lines of code contributed to a substantial cost reduction. Therefore, from the perspective of cutting operational costs having the capacity to capture inefficiencies occurring in the software portfolio's code is vital. For the MIS class of software code optimizations in the area of database interaction loom as particularly worth extending the efforts.

The reality of large organizations is such that mainframe usage monitoring is not implemented in a structural way which allows tracking CPU resource consumption and translating it into management action plans. All this takes place in the face of presence of vastly available tools that support SQL performance monitoring. IBM manuals alone describe many potential ways to assess DB2 performance including DB2 Optimizer, SQL EXPLAIN statement, OS RMF and SMF data, OMEGAMON, RUNSTATS, Materialized Query Tables, Automatic Query Rewrite, or Filter Factors. There are also dozens of third party products that measure DB2 performance. While these tools are helpful in code optimizations leading to lowering CPU resource consumption whether or not they can be used depends on the actual circumstances in which the software runs, or simply on its intrinsic nature. In our research it has turned out that due to business constraints we were unable to use any of these monitoring tools.

1.2. Business reality

Our case study involves a large mainframe production environment comprising 246 Cobol systems which propel an organization operating in the financial sector. The underlying source code contains 23,004 Cobol programs with over 19.7 millions of physical lines of code. Approximately 25% of the programs interacts with DB2. The portfolio spans decades with the oldest program dating back to 1967. Some programs, written in the 70s, are still being altered and used in production. The development environment for this portfolio resembles a technology melting pot as it is highly diversified. The code is quite diverse. Some is hand written while at least five different code generators were also used, all compiled using four different compilers. These characteristics clearly exhibit that executives deal with a complex IT entity. And, assuring operational risk management in such a context is a non-trivial task.

The portfolio serves millions of clients worldwide in various domains including retail, investment and corporate services. The portfolio is business critical; if any of the 264 systems fail then the business might not be able to carry on. The applications must meet stringent up-time and performance requirements. Any action that might endanger operational continuity is unacceptable. Alterations to the production environment are naturally possible but are avoided unless they are strictly necessary. Other than operational risk the managers must also bear in mind a large effort involved, for instance, in testing, validating and releasing. And, while cost cutting is of high importance the requirement for properly functioning software takes precedence.

Constraints Extending efforts to enable portfolio-wide control of CPU resource usage must be fit into the reality in which this business operates. In this context we faced a number of constraints of which two were essential for our choices. One being of a contractual nature. The other concerning operational risk.

The IT-portfolio is maintained by a third party. As a result the organization did not have direct access to the mainframe itself. Gaining access to the machines turned out to be far from trivial under the existing outsourcing agreement. Only a small group of dedicated people from the contractor side had access. Such setup was put in place to allow the contractor almost unrestricted control over the mainframe and enable fulfilling strict conditions stipulated by the service level agreements. In these circumstances we were able to obtain off-line access to mainframe usage reports and source code. Particularly, we had data on the CPU usage consumption in the IMS-DB2 production environment, and the source code, that is it.

Moreover, in this particular portfolio small time delays potentially can have a large impact on the continuity of business operations. In the 246 systems hard coded abnormal terminations of transactions (so called ABENDs) were implemented if certain database operations took too long to operate, like an AICA ABEND. Within this company a very small adaptation of the system time immediately created havoc and led many transactions to be canceled. On one occasion when an engineer set the system time slightly back because of detected deviations between real and system time the hard coded resets fired. Since applying profiling tools might have influence on performance it can also trigger these hard coded resets erroneously. Not a single IT-executive within the firm wanted to take such risks given the significant problems the system time adjustment incident caused.

Naturally, to improve CPU resource consumption one must resort to conducting some sort of measurements. One possibility is to instrument source code with debugging lines to enable, for instance, recording the execution times of particular actions. And, use the obtained measurements to determine which code fragments are likely to be CPU intensive. Obviously, such an endeavor could work if we dealt with several programs still in development but it becomes completely unrealistic for an operational portfolio of 246 systems. By doing so we would have been taking an unknown risk for the production environment. Furthermore, there is a prohibitively high cost involved. In [46] the authors discuss the cost realities of large scale software modifications. Simple single-site releases of business-critical systems easily cost 20 person days, which amounts to 20,000 USD when you take a daily fully burdened rate of 1000 USD. So a release of the 246 systems portfolio, let alone any code changes, can cost 4,920,000 USD ($246 \cdot 20 \cdot 1000 = 4{,}920{,}000$). Clearly such costs are intolerable to executives especially when considering a cost-reduction project at low-risk.

In our research we had the opportunity to analyze a portfolio of 246 applications on top of which a large financial institution operates. Changing the code was an evolutionary process: as soon as one of the 246 systems was due for maintenance, also the performance issues were taken into account. Exceptions were candidates that posed serious performance issues. The idea of automated changes and a big bang of recompiling, installing and testing 246 systems simultaneously is hugely expensive and risky whereas our approach was low-risk, low-impact and evolutionary.

Moreover, our case study focuses on online transactions of the DB2 systems. These types of workloads were identified by our customer as crucial in handling day to day operations. It was indispensable for these transactions to complete in reasonable times to assure a manageable load on the servers and, far more important, to cater for an adequate user experience. The client admitted that apart from these workloads the IT department also did oversee a large collection of batch programs. In particular, computationally expensive SQL queries used in the data warehousing context were mentioned. Batch programs, however, were not identified by the managers as bearing significant business concerns at the time of our case study.

Summarizing, in our setting any approach that influences the performance of the applications was out of the question. This is not only due to the hard coded ABENDs that are present but also due to the effort relating to monitoring the execution behavior in such a large IT entity. Clearly, usage patterns of the 246 systems change and in order to monitor them one would need to deploy profiling tools or enable detailed logging (e.g. collecting specialized SMF records). Such actions might contribute to lowering performance and increase usage costs as a result of extra CPU load. So observing the behavior of systems using standard means bears the risk of not being easily able to cover the entire portfolio.

1.3. Goal of the paper

The paper focuses on the analysis of a large software portfolio from the perspective of lowering operating costs through optimization of the underlying source code. In this work we present findings from our analysis of the business which relies on the software, analysis of the source code base, and also analysis of the work carried out by a third party vendor which involved optimization of the DB2 related code with the objective of reducing operating costs.

The goal of this paper is to show how our work led to creating a set of techniques for uncovering SQL source code inefficiencies which can be applied beyond our studied context to a broader set of IT organizations. Specifically, the objectives are to produce a set of techniques that provide:

- Environmental Isolation – The techniques for uncovering SQL source code inefficiencies can run in an analysis environment (the hardware/software where the analysis is carried out) that is physically separate from the target or production environment (the hardware/software where the systems under study run). This allows for the identification of deficient code without monitoring, changing, or even having access to the target (production) environment.
- Platform Independence – The target and analysis environments are able to run on totally diverse platforms; e.g. different hardware types running different operating systems. For example, the target environment could be an IBM zSeries mainframe running under z/OS, while the analysis environment is Unix running on an HP server.
- Extensibility – The techniques are applicable to any SQL based DBMS, running under any operating system, on any hardware. This means that the techniques work equally well analyzing SQL code written for DB2, Oracle, SQL Server, etc.

1.4. A light-weight approach

In this paper we present an approach that gives executives an option to plan a source code improvement project without having to engage many resources and take unnecessary risks. For instance, in the investigated industrial portfolio we found a relatively small number of source files which were likely to be responsible for higher than necessary CPU resource consumption. In fact, these programs constituted approximately 0.5% of all the programs in the portfolio. And, we found these in a single day by combining best practices for DB2, static code analyses, and historical mainframe usage data.

As it turned out our light-weight approach pinpointed the same hot spots for SQL improvements which were identified in an earlier small-scale pilot SQL-tuning project. The pilot encompassed some selected applications supporting operations of a single business unit. It was executed by an expert team specializing in SQL code performance improvements. The longitudinal data showed reduction of 9.8% in annual MIPS related costs for the optimized part of the portfolio. Due to the sensitive nature of the data we dealt with we cannot provide any monetary figures that characterize cost savings resulting from this project, or any other costs involved in operating the studied mainframe. According to the management the value of the estimated savings significantly outweighs the cost of the project.

Since the pilot showed large cost reduction potential it was decided to scale up the SQL related performance improvements to the portfolio level. Executives were convinced that expanding the approach used in the pilot on a much larger portion of the portfolio could imperil the business operations. Therefore, we designed our approach. Our proposition incorporates heuristics for possibly inefficient SQL expressions. We discuss the set of heuristics we proposed and present the results of applying these across the entire portfolio. Moreover, the input from the small-scale SQL-tuning pilot gave us details concerning the actual code changes and their effect on the CPU consumption. We discuss the inefficient code identified by the expert team and the improvements they introduced. We present the results of applying our approach to a much larger portion of the portfolio which span across multiple business units. We analyzed two scenarios for code improvements that were determined on the basis of our approach. Based on the data from the small-scale SQL-tuning pilot we estimated the possible effects on the CPU consumption for both scenarios. Our approach turned out to be fully transparent for business managers; even those with no deep IT background. After writing this paper we found out that our proposition is being applied by others in the industry [21].

In our approach we rely on finding opportunities for improving applications' source code so that CPU usage can be strongly decreased. CPU usage is a major component of mainframe costs, however, it is not the only component. IT savings are only real if CPU cost reduction techniques are not eclipsed by increases in other resource consumers such as disk space, I/O, journaling, logging, back-ups and even human activities. For example, suppose you reduce the CPU cycles by eliminating `ORDER BY` clause in some SQL query at the cost of adding an index. This requires physical I/Os and disk storage. Although it is typical that CPU costs are higher than storage costs, trade-offs of that sort need to be taken into account and evaluated on case by case basis to assure net savings for IT. Therefore, in any code optimization project based on our approach it is advised to consider broader implications so that code alterations have a net positive effect on the overall IT costs. Of course, the actual changes are to be carried out by specialists who take other aspects into consideration, as well.

The approach we present has two advantages. The first advantage is that the analysis does not have to run in the same or similar environment as the production system. Let us take as an example a target environment of an IBM zSeries mainframe, running z/OS and DB2 and an analysis environment of an HP server running Unix or almost anything else. The advantage here is that performing an analysis on a laptop rather than an IBM mainframe is considerably cheaper and, given usage constraints, probably much faster. The second advantage equally significant is the possibility of using the approach on a different target environment. This approach could work whether the target environment is an Oracle application running on an HP server or an SQL Server application running on a Dell computer. The bottom line is that our approach can be used on virtually any SQL based application regardless of underlying hardware or operating system.

1.5. Related work

There is a vast amount of work devoted to database related software engineering. Here let us mention the work of Chris Date [16,15,14], Michael Blaha [8,7,53], Peter Aiken [2] and Kathie Hogshead-Davis [17]. In Europe Jean-Luc Hainaut [18] and Jean Henrard [28,26,27] have worked on the re-engineering of databases to promote performance.

Software engineers have plenty of resources to reach for when working towards optimal code within DB2 environments. IBM alone has a rich collection of technical manuals devoted to writing efficient queries [32,35,30,43,40,34,39,41,33]. There are also articles from industry practitioners with recommendations concerning DB2 SQL queries tuning [55,54].

Research devoted to control of CPU resource usage in mainframe environments is essentially conducted in the commercial sector. In [12] two factors that impact CPU resource consumption are given: the inefficient applications' code and recurring applications' failures. We use these findings to steer our work and therefore in this paper we concentrate on an approach dealing with code monitoring. Many industry surveys and guidelines are available, for instance, in [10–12,20], and provide valuable insights into CPU resource consumption issues relating to DB2. We incorporate this knowledge into the approach we propose and, in addition, we share the insights obtained from analysis of the case study.

The financial institution we studied exists in a world of stringent industry performance requirements. It is therefore imperative that the approach we develop seamlessly reflects and fits in with their industry standards. In [62] the author presents a solution to the problem of time-optimized database queries execution. That university project reached the industry in the form of a commercially distributed database product, known as MonetDB, to enable delivery of IT-systems with improved database response time. Their research focused on speed, ours focused on costs. We, on the other hand, address the omnipresent IT management issue of controlling MIPS attributed costs and strive to provide executives with management tools to manage those costs. Similarly as in [57] we also investigate an industrial portfolio. Our approach is, in fact, fitted into the realities of large organizations. It should be noted that our work reports on a completed and successful analysis project. Not only was our proposition developed on top of an exceptionally large case study but also results were presented and discussed with the executives.

Application of source code analysis to extract information is omnipresent in the literature. In [46] the authors show how source code analysis supports reduction of costs in IT transformation projects. Literature provides examples of its application in supporting recovery of software architecture [3] or migration of the IT-portfolio to a service oriented architecture (SOA) model [24]. There are also numerous instances of automated software modifications [47,59,58] aided with code analysis. In our work we also rely on source code analysis to extract information relevant from the MIPS control perspective. In that aspect our approach is similar to a technique for rapid-system understanding presented in [56]. It turned out that sophisticated parser technology is not necessary to reach our goals. A lexical approach to analysis is accurate enough for our purposes.

1.6. Organization of this paper

This paper is organized as follows: in Section 2 we present CPU usage realities of the mainframe production environment which we used as our case study. We provide fact-based argumentation behind focusing on the improvements in the area of DB2. In Section 3 we embark on the problem of inefficient usage of SQL language in the source code of the applications. We introduce an extraction method for locating potentially inefficient DB2 related code. In Section 4 we show how we identify Cobol modules which host the interesting, from the analysis point of view, code fragments. In Section 5 we discuss the MIPS-reduction project which was carried out by a DB2 expert team and involved SQL code tuning in the IT-portfolio we used as our case study. In Section 6 we present our approach in the setting of the entire IT-portfolio. We apply it to the case study and show how to estimate savings from DB2 code improvements. In Section 7 we examine the practical issues relating to the implementation of our approach within an organization. In Section 8 we discuss our work in the context of vendor management and redundancies in mainframe usage. Finally, in Section 9 we conclude our work and summarize findings.

2. MIPS: cost component

Mainframe usage fees constitute a significant component in the overall cost of ownership. The fees are directly linked to the application workloads deployed on the mainframe. Measurements of the mainframe usage costs typically involve two terms: MIPS and MSUs. Although mainframe usage measures are colloquially called MIPS, and often used as a rule of thumb for cost estimation, the actual measure is expressed by means of MSUs. The two measures function in parallel on the mainframe market and constitute input for mainframe usage pricing models.

In this section we first explain the two measures. Next, we discuss transactions as they constitute the prime subject of our analyses. And finally, we embark on the analysis of the MSU consumption for the IMS production environment in which the MIS applications of the studied IT-portfolio were deployed. We present our findings concerning DB2 usage and MSU consumption. We argue that improvements in the interaction between client applications and DB2 have the potential to yield savings for the organizations. We also emphasize that DB2 related source code alterations are characterized with low-cost and low-risk for the business.

2.1. MIPS and MSU

MIPS was originally used to describe speed of a computer's processor [45, p. 136]. Since MIPS are dependent on the CPU architecture they are hardly useful in comparing the speed of two different CPUs. For instance, multiplication of two numbers takes a different number of CPU instructions when performed on particular mainframe and PC processors. For this reason some computer engineers jocularly dubbed MIPS a *misleading indicators of performance* [42,49]. Despite the fact that nowadays Misleading Indicator of Performance are somewhat arbitrary figures they still find their application in the industry since they play a role in determining usage fees.

MIPS is a measure of processor speed alone and for this reason it has come under fire for its inaccuracy as a measure of how well a system performs. How software executes within a mainframe depends not only on the CPU but also on other factors, such as memory usage or I/O bandwidth. To embrace these extra factors IBM began licensing its software according to MSUs. MSU stands for Million Service Units and expresses the amount of processing work a computer performs in an hour which is measured in millions of z/OS service units [45, p. 136], where z/OS is the operating system on IBM mainframes. MSU is a synthetic metric which superseded MIPS for its accuracy as it embraced aspects such as hardware configuration, memory usage, I/O bandwidth, complexity of the instruction set, etc.

MIPS and MSUs are not independent from each other. Originally, when MSUs were introduced they were comparable to MIPS. One MSU was approximately 6 MIPS [45, p. 136]. This relationship has disappeared over time and nowadays MSUs hardly track consistent with the MIPS. In fact, if they did there would be no real need for them.

The fact that MIPS is, in principle, a CPU speed measure has led to confusion over its use as a measure for mainframe usage. A CA senior vice president, Mark Combs, explains it this way [22]:

> MIPS can't measure the actual consumption of work, while MSUs can. MIPS are also capacity based, meaning that users who pay according to MIPS are often paying for capacity they don't need. With MSUs, users can choose capacity- or consumption-based pricing. Shops that run close to 100% utilization most of the time might go with capacity-based pricing, while those who run only at 40% most of the time would go with consumption based to save money.

Regardless of the chosen pricing model both MIPS and MSUs have a direct financial implication. For the purpose of our study we rely on the fact that either increase in MIPS capacity or accrual of MSUs results in the growth of mainframe usage fees which businesses have to include in their operational costs. In our paper we assume the use of the consumption-based charging model and interpret the available MSU figures as a measure of consumed MSUs. We will consider reduction in MSUs consumed by a particular mainframe executed object as equivalent to the reduction of mainframe usage costs.

2.2. Transactions

A software portfolio is typically partitioned over a number of information systems. Each system implements some functionality which is deemed necessary to support some business operation. To illustrate this, let us consider an IT-portfolio supporting operations of a mobile network provider. Let one of the IT supported business operations be registration of the duration of a phone call made by a particular subscriber. Additionally, let us assume that for implementation of this operation two systems are needed: one providing functionality to handle client data, and another one enabling interaction with the technical layer of the mobile network infrastructure such that calls can be reported. Implementation of a call registration operation involves a number of atomic computer operations each of which is accomplished through functionality provided by one of the information systems available in the portfolio. One would refer to such a bundle of computer operations serving a particular purpose, which is implemented through the available IT infrastructure, as a transaction.

In the portfolio we investigated we dealt with 246 Cobol systems. From discussion with the experts it became known to us that IT-systems in the portfolio follow the SOA model. In a portfolio which adheres to the SOA model certain systems are meant to provide functionality in a form of shared services. These services are then used by other systems in order to implement a specific operation, such as a transaction. On mainframes transactions can be instantiated through, so called, IMS transactions. IMS stands for Information Management System and is both a transaction manager and a database manager for z/OS [19]. The system is manufactured by IBM and has been used in the industry since 1969. Mainframe usage resulting from code execution is reported through MSUs. In order to keep track of the processor usage for each execution of a transaction the number of consumed MSUs is measured and reported. A collection of these reports provides a repository

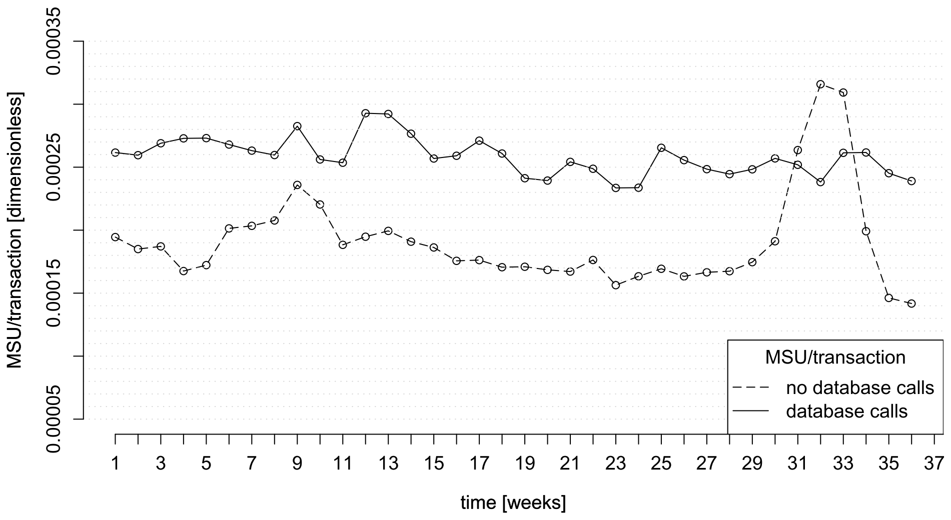

Fig. 1. Time series of the average weekly ratios of MSUs to transaction volumes for two groups of transactions: those which trigger database calls and those which do not.

which enables embracing the overall MSU consumption incurred by the IMS transactions in the IT-portfolio. We used this pool of data as one of the essential inputs in our analyses.

2.3. Database impact

The database is one of the components of the environment in which information systems operate. For mainframe environments the most frequently encountered database engine is DB2. DB2 is IBM's software product which belongs to a family of relational database management systems. DB2 frequently refers to the DB2 Enterprise Server Edition which in the mainframe environments typically runs on z/OS servers. Although DB2 was initially introduced for mainframes [19], it has gained wider popularity since its implementation also exist for personal computers (Express-C edition) [29].

In the context of MIS systems most written programs are client applications for DB2. The SQL language is the common medium to access the database. Interaction with DB2 is, in particular, part of the IMS transactions. This is due to one of the characteristics of IMS transactions. They aggregate computer operations to accomplish some complex task. And, a DB2 operation is one of the many possible operations performed on the mainframes. DB2 utilization is known for being a resource intensive operation. Given that the use of CPU cycles has effect on the associated mainframe usage fees the SQL code run on the mainframes should be, in principle, optimized towards CPU cycles utilization.

MSU consumption In order to get insight into how the database usage participates in the MSU consumption in the studied IMS production environment we analyzed the available mainframe usage reports. We had at our disposal characteristics concerning the top 100 most executed IMS transactions. The characteristics formed time series in which observations were measured on a weekly basis for each top-ranking transaction. For each reported transaction the following data was available: the total number of executions (volume), the total number of consumed MSUs and database calls made. The time series covered a consecutive period of 37 weeks.

For our analysis we distinguished two groups of IMS transactions: those which trigger calls to the database and those which do not. We carried out a comparison of the average cost of executing the IMS transactions belonging to the two groups. We expressed the cost of execution as the average number of MSUs consumed per transaction measured on a weekly basis. In order to make a clear cut between the transactions which make database calls and those which do not we used the numbers of database calls reported for each transaction. We then computed the average weekly ratios of MSUs to transaction volumes for both groups. This way we formed two time series which we analyzed.

In Fig. 1 we present two plots of the time series. The horizontal axis is used to express time in weeks. Each tick indicates a week number. The vertical axis is used to present the ratios of the average weekly MSUs to transaction volumes. The range of values covered by the vertical axis is restricted to the values present in both of the time series. The solid line is a plot of the time series constructed of the ratios for IMS transactions which made calls to the database. The dashed line shows the ratios for transactions which did not make calls to the database.

Analysis of the ratios reveals that the average number of MSUs required to execute an IMS transaction differs between the groups. In Fig. 1 this fact is clearly visible by the relative position of the time series plots. Nearly during the entire time the ratios expressing the average MSU consumption by the transactions not using the database are below the other ratios. Only between weeks 31 and 33 we see that these ratios are above those expressing the average MSU consumption by the database using transactions. We found this case interesting and examined closer the MSU figures for rankings covering the

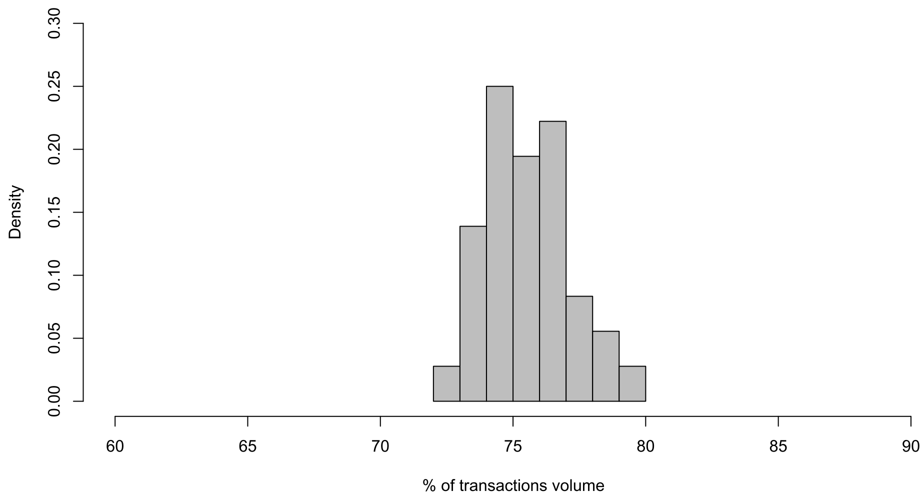

Fig. 2. Distribution of the percentages of the total weekly aggregated transaction volumes of the database interacting IMS transactions.

period between weeks 31 and 33. Our analysis revealed one IMS transaction which had an exceptionally higher than usual MSU consumption reported in those weeks. For this transaction the average weekly number of MSUs consumed in the period from week 1 until 30 was approximately 1299.227. In weeks 31 through 33 the reported consumptions were 5327.130, 8448.990, and 7022.570, respectively. In each case these values were at least four times the average usage between weeks 1 and 30. We did not have enough information to investigate why the temporary peaks occurred. We suspect that one reason could be some non-optimal change to the transaction's implementation and its migration to the production environment. We noticed that the transaction did not occur in the top 100 rankings for the weeks 35 and 36. This might suggest its removal from the production environment.

We compared how the average weekly MSU consumption by the IMS transactions from the two groups differs. We took the long-term average of the ratios for both of the time series. For the transactions which trigger calls to the database the average was 0.0002573. For the transactions which do not, 0.0001914. After we removed the outlier, the transaction which caused the peak between weeks 31 and 33, the average number of MSUs consumed to process an IMS transaction which does not trigger calls to the database came down to 0.000158. These calculations clearly show that on the average the database interacting transactions are more MSU intensive than those which do not interact with the database. Considering the computed averages we observe a difference by nearly as much as 63%.

Database importance The single fact that IMS transactions that perform database calls use more resources than transactions that do not is in itself not a proof that calls to the database are responsible for the major resource usage. While it is likely a more thorough analysis of the proportion of MSUs consumed as a result of executing the Cobol's object code and the MSUs related to DB2 operations within the transactions is necessary. Otherwise, there are a number of other possibilities, for example, it could be the case that the database-involving transactions are simply more complicated than the non-database transactions. Nevertheless, as it turned out the importance of database in the production environment is significant. To investigate this we analyzed the proportion of the database interacting transactions volume in time. Again, we considered the data from the weekly top 100 rankings. For each week we totaled the reported aggregate transaction volumes of those transactions which triggered calls to the database. We then computed the percentages of the total transaction volumes in each week.

In Fig. 2 we present the distribution of the percentages of the total weekly aggregated transaction volumes of the database interacting IMS transactions. We restricted the range of values presented on the horizontal axis to 60% through 90% in order to clearly present the distributions. We did not find any percentages in the data sample outside this range. As we see in the histogram the bars are concentrated to the middle part of the plot. This clearly exhibits that the transactions interacting with the database occupy the majority of the top executed transactions. A great portion of the core business applications which code we inspected are used to handle customer data which are stored in the database. So, our findings are in line with that.

2.4. CPU usage sources

There are all kinds of CPU usage sources. For instance, screen manipulations, algorithms characterized by high computational complexity, expensive database queries, etc. When facing the task of reducing CPU resource consumption any of these elements is a potential candidate for optimization. However, from the perspective of maintenance of a business critical portfolio it is unlikely that an executive is interested in a solution that could imperil operations of a functioning IT-portfolio. It is commonly desired that an approach embodies two properties: low-risk and low-cost. By choosing to improve interaction with DB2 at source code level it is possible to deliver these two properties.

Improvements done in the area of DB2 are low-risk. By low-risk improvements we mean software maintenance actions which do not introduce significant changes to the IT-portfolio. Especially with regard to source code. Our approach does not encourage a major overhaul. As we will show in most cases minor modifications in a few lines of code or configuration changes in the database are sufficient to achieve changes in the MSU consumption. This small scope of alterations is partly due to the fact that DB2 engine provides a wide range of mechanisms which allow for affecting the performance of execution of the arriving database requests. Also, the SQL language allows for semantic equivalence. This opens vast possibilities to seek for other, potentially more efficient, expressions in the source code than the existing code. Due to the fact that database execution performance improvement deals with relatively small changes usually little labor is required. This makes the DB2 related improvements low-cost approach to reducing CPU resource consumption.

Based on the analyzed mainframe usage data we have found evidence that in terms of the average number of MSUs consumed the database interacting IMS transactions cost more than other transactions. Also, these transactions are among those most commonly executed. These observations suggest that by embarking on improvements of the DB2 interacting transactions we address a meaningful cost component on the mainframe.

3. DB2 bottlenecks

In this section we focus on communication between DB2 and the client applications. First, we present what factors impact performance of DB2. In particular, we concentrate our attention on the way SQL code is written. Next, we show cases of inefficient SQL constructs and propose a set of source code checking rules. The rules are syntax based and allow for isolation of code fragments which bear the potential to hamper the CPU when processed by the database engine. Finally, we discuss how we implemented the source code checking process to facilitate the analysis.

3.1. Performance

Performance of DB2 depends on various factors such as index definitions, access paths, or query structure, to name a few. A DB2 database engine provides administrators and programmers with a number of facilities which allow to influence these factors [35]. However, these facilities require accessing the production environment. When approaching reduction of CPU resource consumption from source code perspective it becomes necessary to examine the SQL code. Therefore in our approach we employ DB2 code analysis to seek for possible improvement opportunities.

The way one writes SQL code has a potential to significantly impact performance of execution of requests sent to DB2. This phenomenon is not different from how performance of the execution of programs written in other programming languages is affected by coding style. Virtually any code fragment is inevitably destined to perform inefficiently when inappropriate language constructs or algorithms are used. Writing efficient SQL code requires extensive experience from the programmers, solid knowledge of the language constructs, and also familiarity with the mechanics inside a database engine. In most cases following recommendations of experienced DB2 programmers and fundamental SQL programming guidelines allows obtaining code which runs efficiently. Even though it is the functional code that the consumers are after these days code efficiency cannot be neglected. This is particularly important in the face of growing complexities of queries encountered in, for instance, data warehousing applications.

For business owners inefficient SQL code is highly undesired in the IT-portfolio at least from one perspective; it hampers the speed in which operations are accomplished for the customers. And, of course, in case of mainframe environments it costs money since it wastes CPU resources. Even though static analysis of the programming constructs used in SQL is not sufficient to conclude whether the code is inefficient, it certainly leads to finding code which has the potential of being inefficient. Such code is a good candidate for an in-depth analysis, and if determined as inefficient, for optimization. SQL code optimizations involve, for instance, rewriting, changing the manner in which it is executed, its removal in case it turns out to be redundant, or reconfiguration of the database so that the time needed to execute the query is improved. Application of any of these depends on particular instances of the SQL code. The bottom line is that these instances must first be found.

3.2. Playground

We concentrate on the analysis of the use of the SQL language in the source code of the software applications. SQL facilitates support for various data manipulation tasks. It provides for a whole range of operations. For instance, creation of database objects, maintenance and security, or manipulation of data within the objects. SQL's syntax is based on statements. The statements are commonly categorized according to the type of function they perform. Normally the following three categories of the language are distinguished: Data Definition Language (DDL), Data Manipulation Language (DML), and Data Control Language (DCL) [52, p. 18]. Statements classified as DDL allow to create, modify or delete database objects. The DML statements allow for manipulation of the data within objects through operations such as insertion, deletion, update or retrieval. The last subset, DCL, allows controlling who, among the database users, has rights to perform specific operations.

Table 1
Examples of the types of SQL statements encountered in the IT-portfolio used as a case study.

Statement category	Statement example	Instances	Percentages
DCL	GRANT	0	0.00%
DCL	REVOKE	0	0.00%
DDL	CREATE	0	0.00%
DDL	DROP	0	0.00%
DML	SELECT	14 387	68.91%
DML	DELETE	1948	9.33%
DML	INSERT	2147	10.28%
DML	UPDATE	2397	11.48%

Apart from those major three subsets there also exist auxiliary statements which do not fall into any of the above categories. An example of such a statement is the `SET` statement of DB2 which assigns variables with values. Although this statement is typically used inside a body of an SQL's stored procedure it is also encountered used independently. In such cases it typically serves as a means to copy values of the database registers into a program's local variables, also known in the context of embedded SQL as host variables.

The division of SQL into subclasses provides a way to map specific subsets of statements to programmers' tasks. For instance, a programmer writing client applications is very likely to limit the scope of SQL statements to the DML group. From the perspective of business applications, which by their nature are clients of the database, the DML subset constitutes the great majority of SQL's vocabulary used in the programs. In fact, the internal documentation for SQL coding standards, which belongs to the organization that provided us with the data, states that only usage of the `SELECT`, `INSERT`, `DELETE` and `UPDATE` statements is permitted in the code embedded in Cobol programs.

In Table 1 we give examples of the SQL statements for each listed subset and, whenever available, provide numbers of their occurrences we found in the organization's portfolio. In the DML category we list statements which allow querying (`SELECT`), adding (`INSERT`), removing (`DELETE`) or updating (`UPDATE`) the data. Interestingly, the `SELECT` statement is generally claimed to be the most frequently used SQL statement [50]. This claim turns out to be true at least for the source code we studied. Occurrences of the `SELECT` statement account for almost 69% of all DML statements.

In the DDL category we show two examples of statements: `CREATE` and `DROP`. These statements allow for creation and deletion of database objects such as tables, indexes, users, and others, respectively. They hardly ever occur inside the code of client applications since manipulation of database objects is taken care of normally during the database setup process, or on some other occasions. In fact, we found no occurrences of these statements in the IT-portfolio under study.

A similar situation holds for the DCL subset of statements. In Table 1 we give examples of two such statements: `GRANT` and `REVOKE`. They are used to grant rights to users to perform specific operations on database objects or to revoke these rights, respectively. Due to the nature of the operations these statements perform they are used occasionally by database administrators.

In our analysis we concentrate on the SQL code which is embedded in the applications' source code. In particular, we primarily focus on investigating the `SELECT` statements. While it is possible to formulate inefficiency heuristics for other types of statements, the `SELECT` statement offers a highly desired feature. It is characterized by a complex structure which offers vast capabilities to code queries in numerous ways so that semantic equivalence can be preserved. Obviously, this property is very much sought after from the code optimization perspective. Other than that, the statement is commonly addressed by the DB2 experts community for inefficiency related problems. And, as it turned out, it is the most frequently occurring statement in the studied portfolio. In SQL code analysis we also concentrate on prohibitive use of SQL code in applications. Let us recall that according to the company's proprietary coding standards the embedded SQL is meant to be limited to DML type of statements. Therefore, when formulating characteristics of the possibly inefficient SQL constructs we look also at these deviations.

3.3. Potentially inefficient constructs

Given an SQL statement at hand we want to analyze its structure and determine whether it carries any signs of being potentially expensive for DB2 processing. Let us emphasize that on the basis of the SQL code we are only able to detect signs of potential inefficiency. Thorough analysis of the actual environment in which the statement is executed allows to determine whether the statement is indeed running inefficiently. For instance, despite the fact that the operation of sorting datasets is in general deemed to be expensive its execution is likely to go unnoticed when performed on a data set comprising a hundred of records. However, it is expected to take considerable amount of time when performed on a data set with over one million records. Nevertheless, analysis of SQL code delivers information which when considered at the level of an IT-portfolio enables a low-cost and reasonably accurate assessment of the amount of code suitable for improvements.

In order to exemplify how analysis of an SQL's statement code brings us to finding potentially inefficient constructs, let us consider a general coding guideline that recommends avoiding the lack of restriction on the columns which are to be fetched as a result of processing a `SELECT` statement passed to DB2 [35, p. 215]. On the level of an SQL query's code lack of restriction on the columns is implemented by the use of the `*` character in the `SELECT` clause of the `SELECT` statement.

Table 2
Programming constructs potentially leading to inefficient use of hardware resources during DB2 processing.

Construct ID	Meaning
AGGF	Aggregate function present in a query. Under the AGGF identifier all aggregate functions are included except for STDDEV, STDDEV_SAMP, VARIANCE, and VARIANCE_SAMP.
$AGGF_2$	SQL query contains one of the following aggregate functions: STDDEV, STDDEV_SAMP, VARIANCE, and VARIANCE_SAMP.
COBF	SQL statement is used to load to a host variable a value obtained from one of the database's special registers.
DIST	The DISTINCT operator is present in a query.
GROUP	Query contains GROUP BY operator.
$JOIN_x$	Join operation present in a query. The subscript x is used to indicate the number of tables joined.
NIN	NOT IN construction applied to a sub-query.
NWHR	Missing WHERE clause in a query.
ORDER	Query contains ORDER BY operator.
UNION	UNION operator is used.
UNSEL	No restriction on the column names in the SELECT clause of a SELECT statement.
$WHRE_x$	WHERE clause contains a predicate which contains host variables and constants only. The x provides the total number of such predicates in the WHERE clause.
$WHRH_x$	WHERE clause contains a predicate which contains host variables and column names. The x provides the total number of such predicates in the WHERE clause.

Let us now consider two semantically identical programs with SELECT statements where one carries an unrestricted and the other a restricted SELECT clause. Semantic equivalence in SQL is possible since as a high level language it enables specification of relational expressions in syntactically different but semantically equivalent ways. In consequence of this a query that obtains the required data from the database has possibly many forms. As an illustration let us assume that there exists a DB2 table called PEOPLE which consists of 50 columns, among which two are named FNAME and LNAME. Suppose also that the programs require for processing a list of all pairs of the FNAME and LNAME that exist in the table. The following are two possible SQL query candidates which allow fetching the required data.

```
1. SELECT * FROM PEOPLE

2. SELECT FNAME, LNAME FROM PEOPLE
```

In the first query the SELECT clause contains the * character. Execution of the query results in fetching all rows from the table with all possible columns defined in the table PEOPLE. In a real-life scenario this table possibly contains hundreds of thousands of rows (e.g. a table with names of clients of a health insurance company). In the second query instead of using the * character the needed column names are listed explicitly between the keywords SELECT and FROM, as per SQL language convention. In this case the database will only return the specified values. Generally, restricting what columns are to be fetched during query execution allows DB2 to retrieve only the needed data and thus constraining usage of the hardware resources to the necessary demand. In a real-life scenario the second query is expected to consume less resources than the first one. And, in case of execution on a mainframe be more efficient cost-wise.

There are more syntax based signs in the SQL statements which allow classifying them as potentially expensive. We used the organization's proprietary guideline on internal SQL coding standards, IBM's recommendations concerning DB2 SQL queries tuning, experts recommendations, and also our experience in order to identify practices which are known to have a negative effect on DB2 performance [32,35,55,54]. Based on these sources of knowledge we determined a set of syntactic rules which point at potentially inefficient SQL statements. The point of the rules was to have the means to check if the coding recommendations are followed. Namely, for a given SQL statement some recommendation is not followed if we recognize that the statement's code complies with some syntactic rule. For instance, the aforementioned recommendation concerning explicit listing of the column names which are to be fetched by a query is violated if in the code of the corresponding SELECT statement there exists a * character in the SELECT clause.

Each syntactic rule is associated with a programming construct. For each such programming construct we assigned an identifier in order to allow for simple reference throughout the paper. We first summarize the selected programming constructs in Table 2, and then treat them in greater detail.

Table 2 presents the list of SQL programming constructs which potentially lead to inefficient use of hardware resources when sent to DB2 for processing. In the first column we list programming constructs identifiers. In the second column we explain how each selected programming construct is detectable at the code level of an SQL statement.

The first two programming constructs listed are AGGF and $AGGF_2$. Both of them are related to the presence of aggregate functions in an SQL query. In SQL, aggregate functions constitute a special category of functions that return a single value which is calculated from values present in a selected table column. SQL provides a number of those functions. The actual set differs depending on the version of the database engine. For instance, the SQL reference manual for DB2 9 for z/OS lists the following functions as aggregate: AVG, COUNT, COUNT_BIG, COVARIANCE, COVARIANCE_SAMP, MAX, MIN, STDDEV, STDDEV_SAMP, SUM, VARIANCE, VARIANCE_SAMP and XMLAGG [33, p. 252]. Previous versions of DB2 support smaller subsets of these functions [43,37]. Evaluation of a query containing an aggregate function is deemed costly if the aggregate function is not used in a manner which enables DB2 to carry out processing efficiently. For most of the functions there exist conditions that must be satisfied to allow for evaluation with minimal processing overhead [32, p. 753]. Based on the SQL manual we split the aggregate functions into two groups, namely those functions for which there exist conditions which when satisfied enable DB2 to evaluate efficiently and those for which evaluation is costly regardless of the way the aggregate functions are used. The latter group is formed by the four functions: `STDDEV`, `STDDEV_SAMP`, `VARIANCE`, `VARIANCE_SAMP`, and identified by AGGF_2. For the remaining functions, identified by AGGF, there exist conditions under which cost-effective evaluation is feasible. Since verification on the syntax level whether the conditions are met requires not only full parsing of a query but also additional information on the tables and defined indexes for our purposes we restrict code analysis only to reporting existence of an aggregate function in a query.

Another programming construct we list in Table 2 is COBF. COBF refers to the redundant use of SQL in Cobol programs. By redundant use we mean those inclusions of the SQL code which are easily replaceable by semantically equivalent sets of instructions written in Cobol. An example of it is a situation when SQL code is used to access the DB2's special registers with the pure intention of copying their values into host variables of the Cobol program. DB2 provides storage areas, referred to as special registers, which are defined for an application process by the database manager. These registers are used to store various information, such as current time, date, timezone, etc., which can be referenced inside SQL statements [37, p. 101]. Four of these special registers involve information which is retrievable through an invocation of the built-in, or also referred to as intrinsic, Cobol function called `CURRENT-DATE`. For the following special registers `CURRENT DATE`, `CURRENT TIME`, `CURRENT TIMESTAMP` and `CURRENT TIMEZONE` the stored values are retrievable through that function. Let us recall that in the studied portfolio occurrences of non-DML statements were not permitted. From the CPU resource usage perspective while it is not clear that replacing any SQL code by semantically equivalent Cobol code guarantees reduction in CPU resources usage it is still worth while considering such code alteration. Especially, in cases when relatively non-complex operations, such as current date retrieval, are performed. Therefore we chose to include COBF construct on the list of possibly inefficient programming constructs.

One of the main CPU intensive database operations is sorting. Sorting is a common operation in data processing. SQL provides a number of programming constructs designed to be used as part of the `SELECT` statement which increase the probability that the database will perform sorting at some stage of query's execution. An obvious candidate is the `ORDER BY` clause which literally tells DB2 to perform sorting of the result set according to columns specified as parameters. Therefore detection of the presence of the `ORDER BY` clause is an immediate signal that there might exist a possibly high load on the CPU. Typically, elimination of sorting, if possible, is attainable through adequate use of indexes. Although, this move is likely to reduce load on the CPU it does affect data storage costs which after all might eclipse savings resulting from MSU consumption. This fact should be kept in mind when introducing code changes. Whereas the `ORDER BY` construct is the explicit way to tell the database that sorting is required there are also other constructs which implicitly increase the likelihood of sorting. These constructs are: `UNION`, `DISTINCT`, `GROUP BY` and the join operation [55,54,35]. In Table 2 they are labeled by UNION, DIST, GROUP, and JOIN_x, respectively. The `UNION` construct is a set sum operator. Whenever DB2 makes a union of two result sets it must eliminate duplicates. This is where sorting may occur. Similarly, when the `DISTINCT` construct is present elimination of duplicates takes place. In fact, presence of the `SELECT DISTINCT` construct typically suggests that the query was not written in an optimal manner. This is because the database is told that after the rows have been fetched duplicate rows must be removed. According to [35, p. 584] the sort operation is also possible for the `GROUP BY` clause, if the join operation is present, or the `WHERE` clause contains a `NOT IN` predicate. We report any occurrences of the sorting related constructs encountered in queries. In case of the join operation we additionally report the total number of tables involved. The number serves as the subscript in the JOIN_x identifier.

The `WHERE` clause is a very important component of the `SELECT` statement. In principle, each SQL query should be restricted by the `WHERE` clause to limit the number of rows in the result set. For this reason we check for the presence of this clause and label any SQL query with NWHR in case the `WHERE` clause is missing. Whenever the `WHERE` clause is used it is important that the predicates it contains are structured and used in an manner which enables DB2 to evaluate them efficiently. There are a number of conditions which govern the structuring of the content of the `WHERE` clause. Most of them require an in-depth analysis of how the predicates are used and structured inside the `WHERE` clause, the configuration of the database, and also require considerations as to the data to which the query is applied to.

We focus on several checks in the `WHERE` clause which already give an indication of the potential overhead. One commonly addressed issue is the ordering and the use of indexes for predicates which contain host variables. In [55,54] the interested reader will find details concerning this subject. In order to capture the opportunity for improvements in that respect we check the `WHERE` clauses for the presence of predicates which contain host variables and each time the total number of those predicates exceeds one we report it with the WHRH_x identifier. The subscript is used to specify the number of occurrences of such predicates in the query.

Similarly as COBF, WHRE_x refers to potentially redundant invocations of SQL statements. This time we look at those predicates in the `WHERE` clause for which evaluation does not involve references to the database. In order to explain this let us consider the following SQL query embedded inside a Cobol program:

```
EXEC SQL
    SELECT c1, c2, c3
    FROM table1
    WHERE :HOST-VAR='Andy'
        AND c1>100
END-EXEC
```

It is a simple SQL query which returns a set of rows with three columns: `c1`, `c2`, and `c3`. The results are obtained from `table1`. The constraint imposed on the result set is given in the `WHERE` clause through a conjunction of two predicates: `:HOST-VAR='Andy'` and `c1>100`. Obviously, the only situation in which this conjunction is true is when both predicates evaluate to true. From the structure of the first predicate we see that its valuation has nothing to do with the values in the columns of the `table1`. The host variable `:HOST-VAR` is tested for equality against a constant `'Andy'`. Only the value of the second predicate depends on the table's content. In this particular case it is possible to avoid invocation of the SQL query by rewriting the existing query and changing the way in which it is embedded inside a program. The following is a semantically equivalent improvement:

```
IF HOST-VAR = 'Andy' THEN
    EXEC SQL
        SELECT c1, c2, c3
        FROM table1
        WHERE c1>100
    END-EXEC
END-IF
```

In the rewritten code fragment the `WHERE` clause from the original query was truncated by removing the `:HOST-VAR='Andy'` predicate. Also, the new query was wrapped into the Cobol's `IF` statement. The removed predicate from the SQL query was used as the predicate for the `IF` statement. In the new situation the query is passed to DB2 for processing only when the Cobol's `IF` statement predicate evaluates to true. Of course, in the presented example the improvement was relatively simple. In real-world SQL statements predicates such as `:HOST-VAR='Andy'` may occur in more complex `WHERE` clauses and their improvements be more sophisticated. Nevertheless, the discussed constructs in the `WHERE` clauses ought to be avoided. The number of occurrences of such constructs within the `WHERE` clause are reported in the subscript of the WHRE$_x$ identifier.

In Table 2, we also listed the UNSEL construct which corresponds to the use of unrestricted `SELECT` clause. This construct follows the general recommendation found in [35, p. 215] and was explained earlier in the paper.

3.4. Getting the data

In order to conduct analysis of the SQL code embedded in the source code of a large software portfolio an automated process is a necessity. Let us recall that we did not have access to the mainframe and therefore we could not easily rely on the facilities offered by z/OS to carry out our analysis. Due to the fact that the source code was made available to us on a unix machine we considered the use of tools provided by that platform.

To detect whether an SQL statement contains any of the potentially inefficient SQL constructs listed in Table 2 we must examine the SQL code. To accomplish this task there are essentially two venues to consider: parsing or lexical scanning. The parsing option embodies SQL parser selection, SQL statements parsing, and slicing through the resulting code to check whether the sought after SQL constructs are present. With lexical scanning it is necessary to construct regular expressions capable of matching the desired constructs within an SQL statement. Due to the fact that the SQL constructs listed in Table 2 followed simple textual patterns it was sufficient to employ regular expressions to detect them within SQL statements. To do this we required a technology that is suited to process efficiently a large volume of source files. Let us recall that we dealt with a software portfolio of a large size (23,004 Cobol programs that implement 246 applications). Therefore we opted for incorporating *Perl*. *Perl* is a programming language (and a tool) primarily meant for text processing [60,61]. In particular, it is well suited for a light-weight analysis of the source code. It performs well when processing sheer amounts of data. *Perl* contains a strong regular expressions processing engine and therefore it is adequate to accomplish our tasks. In this way we allowed for an inexpensive, highly scalable, and simple solution to facilitate our analysis.

Tooling In order to accomplish SQL code analysis we distinguish two phases. First, isolation of the SQL code from the applications' sources. Second, verification of the extracted SQL statements against the presence of the programming constructs listed in Table 2. To facilitate these phases we developed a toolset comprising two *Perl* scripts. To illustrate the low complexity of our toolset in Table 3 we provide some of its key characteristics.

In Table 3 we provide characteristics of the developed toolset used to examine the SQL code. In the first column we list the scripts by filename and in the second we explain their role. In columns 3 and 4 we provide the total number of lines of code and the estimated time it took to develop them, respectively. Clearly the 470 lines of code represent a relatively small amount of *Perl* code. The scripts were delivered in as little as 12 hours. That included design, implementation and testing. Our toolset sufficed to obtain the data required for our analysis.

Table 3
Characteristics of the developed toolset used to examine the SQL code.

Script	Role	LOC	Development time
sql-explorer.pl	Used to scan the contents of the Cobol programs and extract the embedded SQL code.	121	4 hours
sql-analyzer.pl	Used to analyze the extracted SQL code. Particularly, to determine the presence of the potentially inefficient SQL constructs listed in Table 2.	349	8 hours
Total		470	12 hours

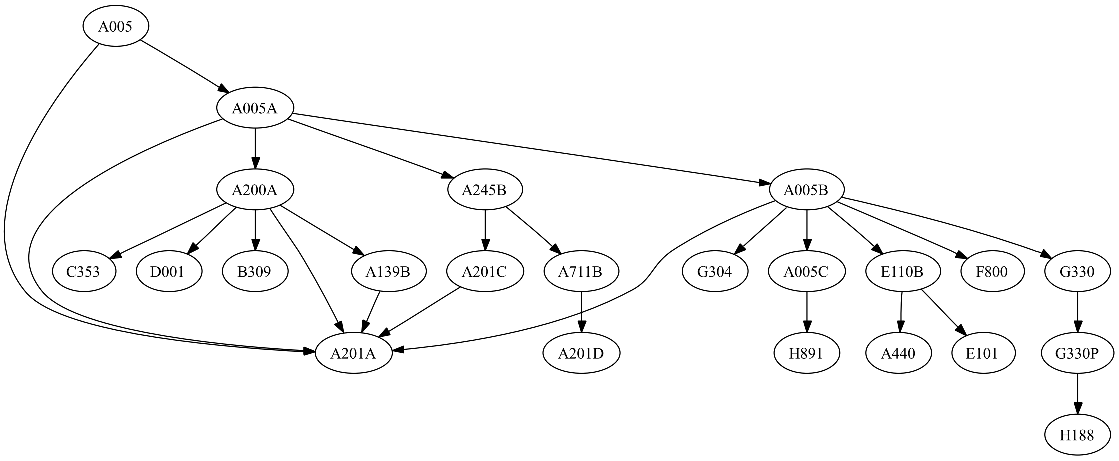

Fig. 3. Call-graph of an IMS transaction.

4. Reaching for code

In approaching CPU usage reduction from source code perspective we must reach for relevant source files. In this section we elaborate on the demarcation of the applicable source files given essential information on the IMS environment. First, we will explain how the mapping between the IMS transactions and their implementation is organized. Next, we discuss the process of analyzing Cobol code which leads to the extraction of data required to identify the relevant source files. Finally, we present properties found for the source code associated with the IMS transactions from the studied IT-portfolio.

4.1. Implementation of transactions

IMS transactions bound to a particular mainframe environment are typically identified by some name. In the studied IT-portfolio the IMS transactions were identified by the names of Cobol programs which served as starting points in the execution of transactions. The source code of the Cobol program which identifies a transaction commonly represents, however, only a small part of the implementation of the transaction. What is typical for Cobol environments, but also encountered in other programming languages, is that a single program invokes a number of other programs when it is executed. Those relationships among programs are known as call dependencies. For this reason in order to find source modules which implement a given IMS transaction it is essential to carry out a call dependency analysis.

Formally speaking, implementation of an IMS transaction is best represented by a directed graph $T_{\mathrm{IMS}} = G\ \langle \mathrm{V}, \mathrm{E} \rangle$. In the context of an IMS transaction V denotes the set of nodes which represent Cobol modules. E denotes the set of edges which represent call relations between the Cobol modules. The edges are characterized by a direction since a call relation always originates at one Cobol module and leads to another. We refer to the T_{IMS} graph as a call-graph.

Fig. 3 depicts a call-graph of one of the IMS transactions taken from the portfolio under study. The plot of the graph was prepared by means of the `dot` program which is part of the Graphviz package, an open source graph visualization software [23]. The data required to feed the `dot` program to construct the graph originated from our analysis of the portfolio's source code. Nodes in the graph represent Cobol programs. The edges of the graph, depicted by means of arrows connecting pairs of different nodes, indicate call dependencies amongst Cobol programs. The direction of the arrow indicates how two modules depend on one another. An arrow originating from node A and reaching node B indicates that a Cobol program (A) contains a reference to a Cobol program (B). In Fig. 3 the labels conform to a pattern: a letter followed by a number, and an optional letter in the end. The letters in the first position are used to associate Cobol modules with IT-systems which they implement. The remaining characters are used to identify different modules within the systems. In the graph presented in Fig. 3 we see call-graph of an IMS transaction `A005`. The node `A005` represents the Cobol module which serves as the starting point in the transaction's execution. At runtime executables associated with the Cobol modules

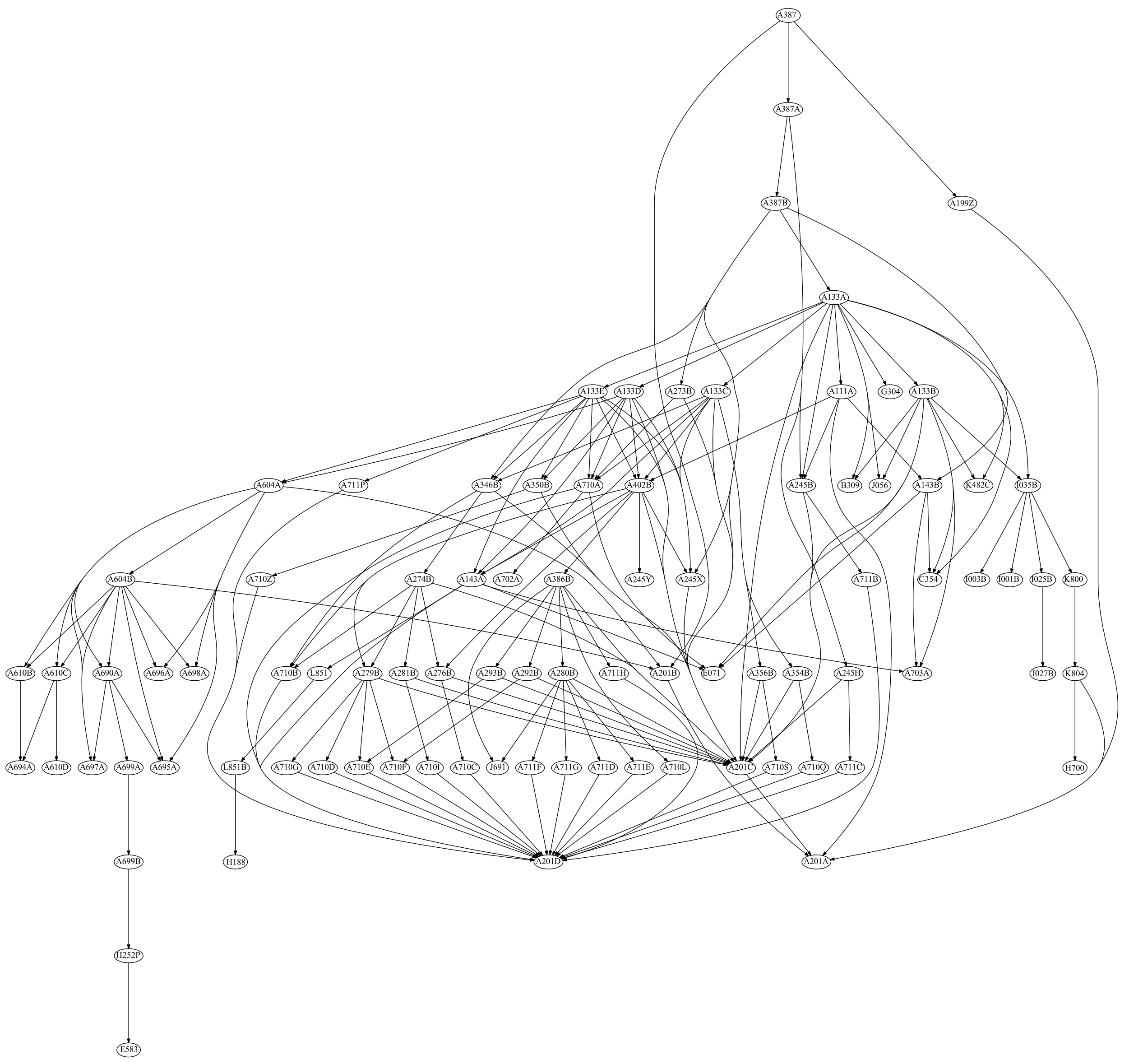

Fig. 4. Call-graph of one of the most executed IMS transactions found in the studied production environment.

represented by the nodes, which appear in the call-graph, have the potential to be executed. Their execution is governed by the program's logic and the flow of data. By analyzing the labels of the nodes in the call-graph it is also apparent that the implementation of the presented IMS transaction spans across 8 different IT-systems.

In a production environment it is not unusual to encounter IMS transactions which involve large numbers of Cobol programs. In Fig. 4 we present a call-graph of one of the most executed IMS transactions found in the studied production environment. It contains 89 nodes what corresponds to 89 different Cobol programs. The programs are part of 15 IT-systems. Given the large number of nodes and relations in the call-graph in Fig. 4 it is obvious that it is infeasible to manually analyze the graph. Similarly, as in case of SQL code analysis, in order to determine the relevant Cobol modules we had no choice but to rely on automation.

4.2. Portfolio exploration

Reaching for the implementation of the IMS transactions boils down to carrying out source code dependency analysis. In our context we operated at the level of a portfolio of applications, in particular, the IMS transactions. The granularity level at which we explored the dependencies in the source code was limited to the individual source files. Specifically, we were interested in the call relations amongst the Cobol modules.

In dependency analysis two types of dependencies are distinguished: static and dynamic. Static dependencies between two source modules arise when one module contains in its code an explicitly defined call to the other module. Dynamic dependencies arise when a reference to a module is established at runtime. On the code level this typically means that the control-flow is determined on the basis of the value of some variable.

We restricted ourselves to finding those call relations which are retrievable from source code. With the kind of access to the portfolio we had we chose to analyze source code using lexical means. Such approach bears certain limitations. In principle, lexical analysis does not allow for the full exploration of dependencies established at runtime. To do this one would require construction of a control-flow graph for each studied transaction in order to explore all possible paths traversable during execution. Accomplishing this demands, however, access to the environment in which the transactions are deployed. Given the constrains under which we operated we chose to limit the relevant source files for SQL analysis to those which are retrievable in a lexical manner.

In the studied Cobol sources we distinguish two types of source files: modules and include files. The latter type of files is also known as the copybooks. In our call-dependencies exploration we analyzed the code of the Cobol modules and ignored the copybooks. In the Cobol programming language copybooks are typically used to store data definitions. Their essential role is to allow for simple sharing of the data definitions among various Cobol applications. Given the context of our work we are only interested in locating the source files which contain the embedded SQL. Since it is not anticipated from the copybooks to contain either the executable Cobol statements or the embedded SQL we decided not to involve them in the analysis.

With the taken approach we were able to capture the static code dependencies. Also, we were able to extract certain potential dynamic dependencies. This was possible by capturing the names of the variables, whenever they were used in the implementation of the calls, and retrieving from the code values of the string constants assigned to them. We treated the values as names of the potentially called modules.

Cobol dependencies In the Cobol programming language implementation of call relations is done by means of two statements: `CALL` and `COPY` [48]. The `CALL` statement transfers control from one object program to another within the run unit [36]. The statement has an elaborate structure that allows for inclusion of various parameters which affect the way the call is made, the result returned, and the exceptions handled. The only mandatory parameter that the `CALL` statement requires is a reference to the program to be called. There are two possibilities to provide this reference, either through a literal or an identifier. In the first case the literal is simply the name of another Cobol program which is embraced by the `'` characters. Cobol calls which are implemented in this way are referred to as *static* since at the compilation time it is known upfront what Cobol programs must be available to run the compiled program. The other way to provide reference to a Cobol program in the `CALL` statement is through an identifier. The identifier holds a name of some variable which is defined in the Cobol program. The variable is expected, at a certain point during execution of the program's code, to hold the name of the Cobol program to be called. This type of `CALL` statement usage implements what is known as a dynamic call.

The other statement, `COPY`, is a library statement that places prewritten text in a Cobol compilation unit [36]. The prewritten text is included in a text file (a copybook). The statement begins with the word `COPY` and ends with a `.` character. In between the `COPY` keyword and the `.` character there is room for parameters. Among the parameters that the `COPY` statement takes there is only one which is mandatory and that is the name of the file to be included. The remaining parameters affect the way in which the statement is processed, in particular, what transformations are done to the text which is included. The effect of processing a `COPY` statement is that the library text associated with the name is copied into the compilation unit, replacing the entire `COPY` statement, beginning with the word `COPY` and ending with a period, inclusive. `COPY` statements are typically used to include into Cobol programs predefined data structures. As explained earlier, we skip copybooks in the process of retrieval of the embedded SQL.

Implementation We approach extraction of call relations on per Cobol module basis. For each scanned module we obtain a set comprising names of the referenced modules. In order to create such a set it is indispensable to analyze the `CALL` statements encountered in the module's code. By extracting the literals from the `CALL` statements we obtain the static call relations. By extracting the identifiers and analyzing assignments of constants to them we obtain names of the modules potentially called. Such retrieval of names provides for approximation of the dynamic call relations. We accomplished the extraction of the call relations from Cobol modules by means of a self-developed Perl script.

Code demarcation The extracted data enabled us to associate each Cobol module with a set containing names of the Cobol modules participating with it in call relations. For the purpose of our work for each selected IMS transaction we are interested in constructing a set comprising all the modules which form its implementation. By obtaining the set of all call relations for all of the modules in the portfolio we have enough data to analyze the call-graph of any transaction in the portfolio.

Given a specific IMS transaction we begin analysis of its call-graph by finding the Cobol module which implements the starting point for the transaction. Let us refer to this module as the root module. Starting from the root module we explore all the reachable modules on the basis of the call relations. This process is called a graph traversal. Graph traversal is typically done using Breadth-First Search (BFS) or Depth-First Search (DFS) algorithms [13]. Both algorithms begin at some

Table 4
Distribution of the fan-in metric for the set of Cobol modules implementing the top 100 IMS transactions.

Fan-in [#]	Frequency [# of modules]	Relative frequency
0	100	0.13021
1	447	0.58203
2	109	0.14193
3	44	0.05729
4	13	0.01693
5	15	0.01953
6..10	20	0.02604
11..15	11	0.01432
16..25	4	0.00521
26..50	1	0.00130
51..55	1	0.00130
56..60	0	0.00000
61..70	1	0.00130
71..76	0	0.00000
77	1	0.00130
78	1	0.00130
>78	0	0.00000
Total	768	1.00000

chosen root node and explore all the reachable nodes. As a result they produce a list of visited nodes. We chose DFS since it was best suited for our internal representation of the call relations data. Applying the graph traversal procedure to the main modules of the IMS transactions enabled us to obtain the source files relevant for SQL code analysis.

4.3. Code properties

Call dependencies are important when approaching CPU usage reduction through code improvements. Naturally, alterations to the source code of programs which are dependencies for other programs have impact not only on the overall functionality but also the execution performance. The impact that changes to an individual source file have on the performance depends on the frequency of executions of its object code. When considering a call-graph of an application it is clear that improvements done to the code of those modules which object code is frequently executed have higher impact on the performance than alterations made to the modules which object code is executed less frequently. For the purpose of selecting a possibly small set of modules in which code improvements can allow attaining a significant reduction in CPU usage having the information on the number of executions of individual programs is in principle desired. However, obtaining such information requires a meticulous analysis of the environment in which the programs are executed. Given the constraints under which our approach was applied we had to resort to reaching for an alternative information.

In an effort to obtain some approximation to the information on the frequency of program executions we examined call relationships among the source files. This way we were able to get insight into the intensity of reusability of individual programs in the implementations of the applications. We captured the intensity by counting the number of references to the modules. Clearly, by considering the number of references instead of the number of executions we allow for imprecision. For instance, we might deem as a good candidate for optimization a source code module which is referenced by 100 programs each of which only executes its object code one time per day. Nevertheless, with our goal to support CPU usage reduction through code improvements in a large portfolio this information gives us some indication of importance of the module in the execution sequence. And, with the constrains under which our approach is deployed the information is obtainable without much effort.

Source code analysis resulted in each module being associated with a set of names of modules which it references. Having this data at our disposal we carried out two analyses. The first analysis dealt with investigation of the distribution of the number of call relations of the modules. The second dealt with measuring the relationship between the modules altered and the number of IMS transactions these alterations affect.

Modules fan-in In the first analysis we studied the distribution of the number of references made to modules. In software engineering this number is commonly referred to as fan-in and used as a structural metric. Our analysis of the distribution of the fan-in metric was restricted to those Cobol modules which implement the top 100 most executed IMS transactions. Following the code demarcation process each IMS transaction was associated with a set of modules. Performing the union operation on these sets resulted in isolating 768 distinct Cobol modules. For those modules we determined the fan-in metric. The modules counted as references were limited to the pool of the selected modules. This way we inspected the distribution of the fan-in within a particular group of IMS transactions.

Table 4 presents the distribution of the fan-in metric for the set of Cobol modules implementing the top 100 IMS transactions. In the first column we list the classes for the values of the fan-in metric. In the second and third column we provide the frequencies of module occurrences in each class and the relative frequencies, respectively. Out of the 768 Cobol

Table 5
Distribution of the modules among the IMS transactions.

IMS transactions [#]	Frequency [# of modules]	Relative frequency
1	409	0.53255
2	118	0.15365
3	56	0.07292
4	34	0.04427
5	64	0.08333
6	10	0.01302
7	8	0.01042
8	10	0.01302
9	17	0.02214
10	17	0.02214
11..15	13	0.01693
16..20	7	0.00911
21..25	0	0.00000
26..28	2	0.00260
29..31	2	0.00260
32	1	0.00130
>32	0	0.00000
Total	768	1.00000

modules the value of the fan-in was greater or equal to 1 for 668 of them. The remaining 100 modules were the starting points for the analyzed IMS transactions with a fan-in value of 0.

The fan-in metric values range from 0 until 78. From Table 4 we see that the frequency decays sharply along with the increase in the value of the fan-in metric. We clearly see from the distribution that the vast majority of modules have very few incoming call relations. Those with fan-in metric of value one constitute over 58%. We also find outliers. For the fan-in metric values greater than 26 we find only 5 modules. The top most referenced module has as many as 78 incoming call-relations. The distribution we observe suggests that in the implementation of the top 100 IMS transactions there are only very few modules for which code alterations have impact on a large number of other modules.

Change impact Next to the analysis of the distribution of the fan-in metric among the modules which implement the top 100 IMS transactions we also studied the relationship between the IMS transactions and the modules. Analysis of the fan-in metric revealed to us that in the portfolio we find a group of modules in which each is referenced at least once by some other module. From the CPU resources usage reduction point of view the following question is relevant: how are these modules distributed among the IMS transactions? For each of the 768 modules we counted how many times it occurs in implementations of the IMS transactions. We then analyzed the distribution of the obtained counts.

Table 5 presents the distribution of the module counts among the IMS transactions. In the first column we list the classes with the numbers of IMS transactions affected through alterations to an individual module. In the second and third column we provide the frequencies of module occurrences in each class and the relative frequencies, respectively. The distribution characterized in Table 5 resembles the one in Table 4. In this case we observe that the frequency decays along with the increase in the number of IMS transactions. We find that nearly half of the modules (46.74%) belong to implementations of at least two different IMS transactions. We also find a few outliers. There is one module which when altered has a potential to affect as many as 32 IMS transactions.

The facts revealed through this analysis are meaningful from the perspective of planning a code improvement project. In a scenario in which the goal is to improve performance of IMS transactions it is generally desired to change a few Cobol modules and yet impact as many IMS transactions as possible. Based on the analysis we see that in the portfolio under study there exist many opportunities for choosing modules in such a way that multiple IMS transactions get affected. Exploitation of such opportunities yields ways to seeking configurations of modules which allow maximizing reduction in CPU resources usage while limiting code changes.

5. MIPS-reduction project

A team of experts specializing in optimizations of mainframe related costs carried out an improvement project, dubbed as the MIPS-reduction project, on the source code of the IT-portfolio under study. In this section we present our analysis of the code changes and the impact of the project on the MSU consumption. The objective of the project was to reduce the consumption of MSUs of the selected IMS transactions. The reduction was to be obtained through optimization of the DB2 related fragments in the source code which implement functionality of the transactions. The project was evaluated by comparing the average ratios of MSUs to transaction volumes before and after the changes were made. Our analysis shows that in the course of the project relatively innocent looking changes were made to either the code of the Cobol modules involved in the inspected IMS transactions or to the database configuration, e.g. index modification. Nevertheless, these changes were sufficient to cause a noticeable decrease in the ratio of MSUs to transactions volume. This effect was

Table 6
Characteristics of the portion of the IT-portfolio's source code covered in the MIPS-reduction project.

IMS transactions inspected	6
Cobol modules covered	271
DB2 modules	87
DB2 modules with SQL statements	81
Modules with the potentially inefficient SQL constructs	65
# of instances of the potentially inefficient SQL constructs	424

observable for both the IMS transactions initially selected for optimization and those impacted by the changes through interdependencies in the code.

5.1. Project scope

The portion of the IT-portfolio's source code covered in the project was determined on the basis of the selected IMS transactions. The expert's selection was limited to the pool of IMS transactions which were part of IT services of a particular business domain in the organization. In the selection process the primary point of departure was the ranking list of the top most executed IMS transactions in the production environment. After restricting the ranking to the relevant IMS transactions the choice fell on those transactions for which the average weekly ratio of the MSUs to transactions volume was the highest. From this ranking five IMS transactions were selected. The experts also found an additional IMS transaction, which was not in the ranking. Due to reported excessive CPU usage and the fact it was part of the business domain, for which the project was commissioned, it was selected for code improvements. All of the six selected IMS transactions were interacting with the production's environment DB2 database.

In Table 6 we provide some characteristics of the portion of the IT-portfolio's source code which was involved in the MIPS-reduction project. For the 6 IMS transactions selected for the project we obtained the list of modules which formed their implementation. This process boiled down to finding the source files associated with the implementations of the transactions. As a result 271 Cobol modules were retrieved. Amid those modules 87 (≈32%) were identified as *DB2 modules* (Cobol programs with embedded DB2 code). It means that in each of those modules at least one `EXEC SQL` code block was present. The total number of modules containing SQL statements which were relevant from the MIPS-reduction project perspective was 81. This represents approximately 0.35% of all Cobol programs (23,004) in the portfolio's source code. In the remaining 6 modules we found only the SQL's `INCLUDE` statements. From the modules relevant for the SQL code analysis we isolated 65 in which at least one potentially inefficient SQL construct was identified. The constructs were identified with the SQL analysis tool, which is part of our framework. In total 424 instances of the potentially inefficient SQL constructs were found. The figure encompasses all the potentially inefficient SQL constructs listed in Table 2.

5.2. Selected modules

The expert team made the selection of the code that underwent improvements in a semi-automated process. The process included analysis of the time spent on execution of the DB2 modules involved in the implementation of the selected IMS transactions. In the course of this analysis 6 Cobol-DB2 programs were selected. We scanned the selected programs to acquire characteristics of the SQL code.

In Table 7 we list in the first column the identifiers of the SQL constructs defined in Table 2. We list only the identifiers of those constructs for which at least one instance was found in the inspected code. In the inspected portion of the portfolio's source code we found no SQL statement which contained AGGF_2, DIST, GROUP, NIN, UNSEL, or UNION programming constructs. The next six columns are used to list instances of each construct encountered in the SQL code of the modules selected for optimizations. In the last two rows we summarize occurrences of the instances of the potentially inefficient programming constructs and the `EXEC SQL` blocks in the modules. From Table 7 it is clearly visible that in every module we find some SQL statement which exhibits the presences of the programming constructs dubbed by us as potentially inefficient. As we will show in all of these modules, except for *Module 1*, at least one of the recognized programming constructs was the direct cause of the triggered improvement actions, which resulted in changes to the code.

5.3. Changes

We now discuss the code changes done to the Cobol modules selected in the MIPS-reduction project by the expert team. We present the situation encountered in the code before and after the alterations were made by the expert team. For most of the modules we illustrate the code changes by providing code snippets taken from the original modules. For confidentiality reasons the variable, column and table names occurring in the snippets were changed.

Module 1 The original module *Module 1* contained three SQL queries embedded inside cursors. In these queries we found occurrences of the AGGF, JOIN_4, ORDER, and a series of WHRH_x programming constructs. The only aggregate function found

Table 7
Characteristics of the SQL statements found in the DB2-Cobol modules which implement IMS transactions selected for the MIPS-reduction project.

Construct ID	Module 1	Module 2	Module 3	Module 4	Module 5	Module 6
AGGF	1	7	2	0	3	0
COBF	0	0	0	2	0	0
$JOIN_2$	0	2	0	0	0	0
$JOIN_4$	1	0	0	0	0	0
ORDER	1	4	2	1	1	0
$WHRE_1$	0	1	0	0	0	0
$WHRE_2$	0	1	0	0	0	0
$WHRE_3$	0	7	2	0	1	0
$WHRE_{4..10}$	0	0	1	0	0	0
$WHRH_2$	1	5	2	0	4	2
$WHRH_3$	1	4	2	3	3	4
$WHRH_4$	1	1	1	0	0	0
$WHRH_{5..10}$	0	3	0	0	0	0
Instances in total	6	35	12	6	12	6
EXEC SQL blocks	8	40	20	12	14	28

was present inside a sub-query of one of the cursors. It was used to return the maximum value. Two of the cursors contained ORDER BY clauses. The expert team discovered that the scenario in which these potentially expensive looking queries were used inside the program's code led to inefficient hardware resource usage. They identified what is known as a *nested cursor situation*. Presence of such a situation means that for two different cursors one is opened inside the other one. In case of the three cursors each time a row was fetched using one of the cursors, one of the two remaining cursors was opened, values were retrieved and the cursors closed. Let us recall that for SELECT statements embedded inside cursors the query's invocation takes places each time a cursor is opened. In the scenario found in the program the nested cursors were opened and closed several times.

The improvement to the encountered situation involved changes to both the queries and the program's logic. According to the experts it was possible to supplant the three queries with a single one and obtain a result equivalent to those in the original program setting. Based on this conclusion all the three cursors were combined to form a single cursor, and the relevant Cobol code fragments were adequately adapted.

Module 2 In the original module *Module 2* we found fifteen SQL queries. Inside the code of the queries we found programming constructs of AGGF, $JOIN_2$, ORDER, and various instances of $WHRE_x$ and $WHRH_x$. After the analysis the experts concluded that the ORDER BY clauses present inside two of the SELECT statements, embedded inside cursors, were the cause of the burden on the CPU. The ORDER BY clauses were removed after the existing cluster index on the table was extended with two additional column names used in the original ORDER BY clauses.

Module 3 In the original module *Module 3* a cursor was declared with the following SQL query:

```
SELECT DOCID        ,
       NUMBER       ,
       DATE_CREATED ,
       CREATED_BY
 FROM  FILEDOCUMENTS
 WHERE NUMBER = :HOST-A
   AND DOCID >= :HOST-B
 ORDER BY DOCID ASC;
```

The ORDER BY clause in this deceptively simple query was identified as a performance hampering element. The solution was two-fold. First, the existing cluster index was extended with the field DOCID. And secondly, the ORDER BY clause was removed from the query.

Module 4 In module *Module 4* the SQL programming constructs of COBF, ORDER, and $WHRH_3$ were found. In this module code changes were triggered by the two instances of the COBF programming construct. In this module there were two EXEC SQL blocks which contained code dealing with the retrieval of a value from the DB2's special register: CURRENT DATE. It is starkly redundant to make DB2 calls for pure retrieval of these values while they are obtainable in a far cheaper way, in terms of CPU usage, through calls to the Cobol intrinsic functions. The expert team made a decision to replace the found EXEC SQL blocks with equivalent Cobol code. The following code fragment illustrates what changes were made to the code in module *Module 4* with respect to one of the two EXEC SQL blocks. Changes concerning the other block were done in an analogous manner.

```
--- CODE ADDED -------------------------------------------------
59:       01  WS-TEMP-DATE-FORMAT.
60:           03  WS-TEMP-YYYY           PIC X(4).
61:           03  WS-TEMP-MM             PIC X(2).
62:           03  WS-TEMP-DD             PIC X(2).
--- CODE ADDED -------------------------------------------------
63:       01  WS-FINAL-DATE-FORMAT.
64:           03  WS-FINAL-DD            PIC X(2).
65:           03  FILLER                 PIC X(1) VALUE ".".
66:           03  WS-FINAL-MM            PIC X(2).
67:           03  FILLER                 PIC X(1) VALUE ".".
68:           03  WS-FINAL-YYYY          PIC X(4).
--- CODE DELETED -----------------------------------------------
339:024000        EXEC SQL
340:024100            SET :DCLSERVICEAUTH.SA-STARTING-DATE
341:024200                = CURRENT DATE
342:024300        END-EXEC
--- CODE ADDED -------------------------------------------------
360:              INITIALIZE WS-TEMP-DATE-FORMAT
361:                         WS-FINAL-DATE-FORMAT
362:              MOVE FUNCTION CURRENT-DATE  (1:8)
363:                                     TO WS-TEMP-DATE-FORMAT
364:              MOVE WS-TEMP-DD            TO WS-FINAL-DD
365:              MOVE WS-TEMP-MM            TO WS-FINAL-MM
366:              MOVE WS-TEMP-YYYY          TO WS-FINAL-YYYY
367:              MOVE WS-FINAL-DATE-FORMAT
368:                TO SA-STARTING-DATE    IN DCLSERVICEAUTH
```

Whereas the above illustrated code transformation is a valid optimization solution it does, however, pose a risk for migration of this code to a newer version of a Cobol compiler. According to [31] the semantics of the CURRENT-DATE function changes in the Enterprise Cobol for z/OS. This will imply that in case of a compiler upgrade additional changes will be inevitable for this module.

Module 5 *Module 5* contained the following SELECT statement:

```
      SELECT  START_DATE
             ,DATE_CREATED
             ,CREATED_BY
             ,DATE_RISK_ASS
             ,RESULT
             ,DATE_REVISION
             ,ISSUE_ID
             ,PROCESS_ID
             ,STATUS
             ,DC_ACTIONCODE
             ,ENTRY_ID
             ,DATE_AUTH
             ,AUTHORISED_BY
        INTO  ....
              ....
        FROM TABLE001
       WHERE NUMBER      = :NUMBER-DB2
         AND START_DATE  =
              ( SELECT MAX(START_DATE)
                  FROM TABLE001
                 WHERE NUMBER = :NUMBER-DB2
                   AND START_DATE < :H-START-DATE-DB2
                   AND ( ( :H-DF-IND-DB2      = '0'  AND
                           STATUS          = '0'  AND
                           DC_ACTIONCODE <> '0'
                         )
                        OR
                         ( :H-DF-IND-DB2      = '1'  AND
                           STATUS IN ('1','9')
                         )
                        OR
                         ( :H-DF-IND-DB2      = '2'  AND
                          ( STATUS        <> '0'  OR
                            DC_ACTIONCODE <> '0')
                         )
                       )
              )
```

In this query we found programming constructs labeled by us earlier as AGGF, ORDER, $WHRE_3$, $WHRH_2$ and $WHRH_3$. The experts focused on the WHERE clause inside the nested sub-query. The particularly interesting fragment of the SQL code is between lines 23 and 36 where three predicates concatenated with OR operators are present. In each OR separated block (lines: 23–26, 28–30, 32–35) a host variable :H-DF-IND-DB2 is compared with different constants. The condition specified in the sub-query's WHERE clause will hold true if and only if the value of the host variable is either 0, 1 or 2. Such use of the above presented query in the code leads to potentially redundant invocation of a call to DB2. Regardless of the actual value held by the host variable :H-DF-IND-DB2 the query will always be executed. To circumvent this situation from happening it is desired to first make appropriate comparisons with the host variable :H-DF-IND-DB2 and only if any of them holds true make a DB2 call to execute the query. Such logic is programmable by means of Cobol conditional statements.

In this module, the presented query was rewritten in such a way that three new queries were introduced. Each new query differed from the original with the WHERE clauses. The comparison of the host variable :H-DF-IND-DB2 with the constants was removed from the new queries. The control over invocation of the queries was embedded into a ladder of Cobol IF statements.

Module 6 In the original *Module 6* six cursors were found each of which contained the WHERE clause. All the WHERE clauses were analyzed and the potentially inefficient programing constructs of type $WHRH_2$ and $WHRH_3$ were found. In five of these cursors the WHERE clauses contained a predicate which involved comparison of a database retrieved value with a country code. In all five cursors this predicate was placed as second in a chain of AND operators. According to the experts inefficient ordering of the predicates in the WHERE clauses was negatively affecting the speed of execution of the query. The country code check, which was carried out as first, was not restricting the dataset well enough and thus hampering the performance. All five cursors had their WHERE clauses rewritten by changing the order in which the predicates are checked. The following code fragment illustrates the conducted transformation.

```
1  --- OLD WHERE CLAUSE------------------------------------------
2   WHERE ( PH_COUNTRY = :H-COUNTRY )
3       AND
4   ( PH_FONETICNAME   = :H-FONETICNAME )
5       AND
6   ( PH_YEAR_OF_BIRTH = :H-YEAR_OF_BIRTH)
7
8  --- NEW WHERE CLAUSE -----------------------------------------
9   WHERE ( PH_FONETICNAME  = :H-FONETICNAME )
10      AND
11  ( PH_COUNTRY            = :H-COUNTRY )
12       AND
13  ( PH_YEAR_OF_BIRTH      = :H-YEAR_OF_BIRTH)
```

The code fragment shown above presents the WHERE clause of one of the five queries found in the Cobol program. In the original situation in all of the queries the (PH_COUNTRY=:H-COUNTRY) predicate was listed as first in the sequence of the AND operators (line 2). After changes to the code were made this predicate was replaced with the second one. In addition to the changes made in the code two indexes were created in the database to satisfy two of the cursors.

5.4. Impact analysis

We now present the analysis of the impact that the MIPS-reduction project had on the IMS transactions in the portfolio. Let us recall, we have observed that modules in the studied IT-portfolio are interdependent on one another. This observation turned out to hold true, in particular, for the Cobol modules which implement the IMS transactions selected for the MIPS-reduction project. Interdependencies in the code cause that the effects of code changes propagate to other parts of the portfolio. And, as explained earlier, this has implications for both the functionality and the execution performance of applications. As a consequence apart from the IMS transactions initially selected for the MIPS-reduction project also other IMS transactions became affected due to the carried out code changes. The expert team identified in total 23 IMS transactions which were affected in the aftermath of the project and used them for assessment of the project's performance.

In Table 8 we present a transactions-modules dependency matrix which shows the relationships among the project affected IMS transactions and the altered Cobol modules. The data required to construct the matrix was obtained by retrieving from source files implementing the transactions. The retrieval was accomplished by means of our tools. The columns correspond to modules changed in the course of the project and are labeled *Module 1* through *Module 6*. The rows, labeled T1 through T23, represent transactions pinpointed for the project assessment. Presence of a module i in the implementation of a transaction j is indicated with the symbol X placed on the intersection of the corresponding column and row.

The first observation which follows from the obtained transactions-modules dependency matrix is that a relatively low number of Cobol programs has a wide range of coverage of the IMS transactions. Also, the range in which the changed modules affect the transactions differs a lot. *Module 6* is only part of implementation of the transaction T18, whereas *Module 2* is part of implementation of nearly all listed transactions except for T11 and T21. This observation suggests that

Table 8
Transactions affected by MIPS-reduction project changes: code based impact analysis.

Transaction	Module 1	Module 2	Module 3	Module 4	Module 5	Module 6
T1		X	X			
T2		X				
T3		X				
T4		X				
T5	X	X	X			
T6	X	X	X			
T7		X				
T8	X	X	X			
T9	X	X		X	X	
T10		X				
T11						
T12		X	X			
T13		X				
T14		X				
T15		X				
T16		X				
T17	X	X				
T18	X	X		X		X
T19		X				
T20	X	X		X	X	
T21	X					
T22	X	X	X			
T23		X				

the *Module 2* turned out to be a shared bottleneck for nearly all affected transactions. One transaction for which our analysis did not reveal any relationship with the changed Cobol modules is T11. We were unable to disclose the reason for the transaction's inclusion in the impact analysis of the MIPS-reduction project. We believe that one of the reasons it was listed, despite the fact no code relationship was discovered, is that the DB2 indexes created or expanded in the course of the project had some effect on the transaction's performance.

5.5. MSU reduction

The primary goal of the project was to reduce MIPS usage fees in the production environment which were linked to execution of the IMS transactions. The performance metric used for evaluation of the project's impact on the MSUs was the ratio of the MSUs consumed to the transactions volume in the given time frame for a given set of IMS transactions. Decrease of the ratio in time was considered meeting the project's goal. Obviously, the higher the decrease the higher the reduction of MSUs, and thus the better the result of the project. The expert team evaluated the actual impact of the MIPS-reduction project by comparing the weekly ratios of the MSUs to transaction volumes in two points in time: before and after the changes were migrated to the production environment. For this evaluation the experts considered the 23 transactions listed in Table 8.

We analyzed the effectiveness of the MIPS-reduction project in a time frame spanning across several weeks before and after the project changes were migrated to the production environment. We did this by studying the behavior of the performance metric in time. We took the available weekly data concerning the top 100 most executed IMS transactions. In the considered time frame 19 out of 23 IMS transactions affected by the MIPS-reduction project were consistently occurring in the weekly rankings. For these 19 transactions we had at our disposal totals for their weekly MSU consumption and transaction volumes. We aggregated these figures to obtain weekly totals for the MSU consumption and transaction volumes for the bundle of the 19 transactions. From these totals we calculated the weekly MSUs to transactions volumes ratios.

Fig. 5 presents the obtained time series of the average weekly ratios of the MSUs to transactions volume for the set of IMS transactions affected by the MIPS-reduction project. The horizontal axis denotes time. Each tick on the axis represents a week number. The time series covers the period of 36 consecutive weeks. The vertical axis is used to express the average weekly ratios of MSUs to transaction volumes. The period in which the MIPS-project evaluation took place is distinguished in the plot by the vertical dashed lines which range from week 14 until 19. From the plot it is clearly visible that between week 14 and 19 the average ratio of the MSUs to transaction volumes decreased. Before week 14 the ratio ranged from 0.000271 until 0.000299. After week 19 the ratio decreased and oscillated between 0.000250 and 0.000271. The expert team estimated that the project reduced the annual MIPS related costs attributed to the optimized IMS transactions by 9.8%. The figure was provided by the expert team. Due to the confidentiality agreement we are unable to provide the monetary value of this reduction. According to the management the value of the estimated savings significantly outweighs the cost of the project.

External influence In order to verify that the reduction in MSUs was solely due to the MIPS-reduction project we checked the portfolio source code for changes. In particular, we scrutinized whether the source code which implements the func-

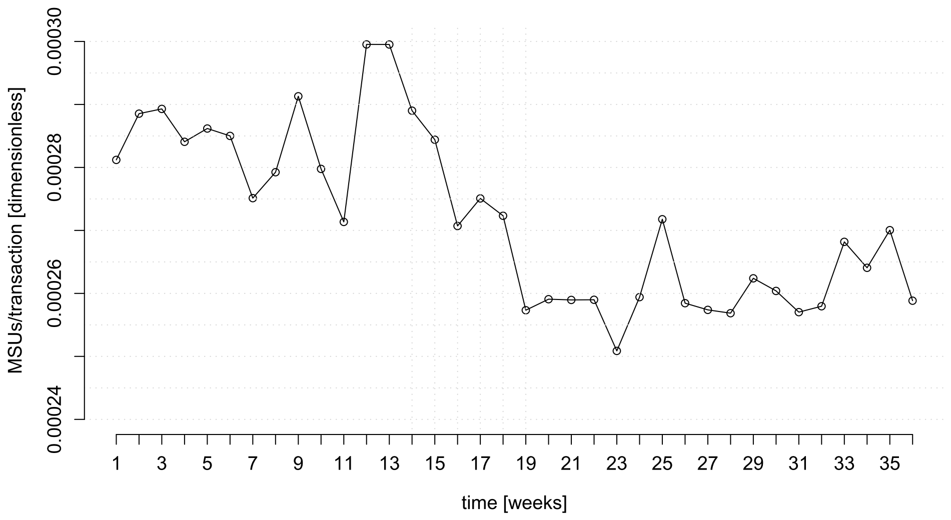

Fig. 5. Time series of the average weekly ratios of the MSUs to transactions volume for the transactions affected by the MIPS-reduction project.

tionality of the affected IMS transactions was altered between weeks 14 and 19. Let us recall, the initial number of Cobol source modules covered by the project was 271. This number was the total number of Cobol modules which implemented the set of the six selected transactions targeted for MSU consumption improvements. In addition to those six transactions the expert team identified other 17 IMS transactions which also became affected by the MIPS-reduction project. To obtain the list of Cobol modules which implemented the 23 IMS transactions we analyzed their call-graphs. In total we found 328 Cobol modules. For each of those modules we looked up in a version control system whether any code alteration were reported and, if altered, migrated to the production environment. The check concerned alterations to source code in the time frame between weeks 14 and 19. Apart from the changes with respect to the modules altered as part of the project we found three other modules which were modified and brought into the production environment in the time. We studied carefully what the nature of modifications in these modules was. Two of the modules contained SQL code but it was not altered. All the modifications we found referred to minor changes in the Cobol code.

Production environment We also studied transaction and database calls volumes in the production environment in the period when performance of the project was measured. The objective of this analysis was to investigate whether there were any observable changes in the volume of transactions or the number of calls made to the database by the IMS transactions before and after project changes were migrated to the production environment. One variable which has effect on the MSU consumption by a transaction is the number of database calls made by the transaction. Decline in the number of calls has a positive effect on the MSUs since it lowers their consumption. For the analysis we considered the time series of the weekly total transaction volumes and the associated total volumes of database calls for the IMS transactions affected by the MIPS-reduction project. The time series covered the same time frame as in the analysis of the average weekly ratios of the MSUs to transaction volumes. Given that the data we had at our disposal were consistently reported for 19 out of 23 transactions we restricted our analysis to those 19 transactions. We aggregated the weekly transaction and database volumes to obtain weekly totals of these properties for a bundle of the 19 transactions. The obtained totals formed two time series which we analyzed.

In Figs. 6 and 7 we present plots of the time series of the total transactions volume and the total volume of database calls made by the IMS transactions affected by the MIPS-reduction project, respectively. In both plots the horizontal axes are used to denote time. Each tick represents a week number. In Fig. 6 the vertical axis denotes the total number of transactions, and in Fig. 7 the total number of database calls. Given that the number of database calls is dependent on the number of transaction invocations it is not surprising that the two plots look alike. In fact, the Pearson's correlation coefficient is high (0.8951222) what confirms the strong correlation. The plots in Figs. 6 and 7 do not exhibit any significant changes neither to the transaction nor to the database calls volume. In fact, the number of database calls is on the rise during the entire time frame.

5.6. Summary

The MIPS-reduction project has shown that changes carried out to the DB2 related code had impact on the MSU consumption. We have observed a decline in the average weekly ratios of the aggregated MSUs and the transaction volumes after the project took place. According to the expert team the estimated annual savings related to the execution of the IMS transactions affected by the project were approximately 9.8%. It turned out that amid the 87 DB2-Cobol programs, which were relevant from the project perspective, changes to only six of them were sufficient to arrive at substantial sav-

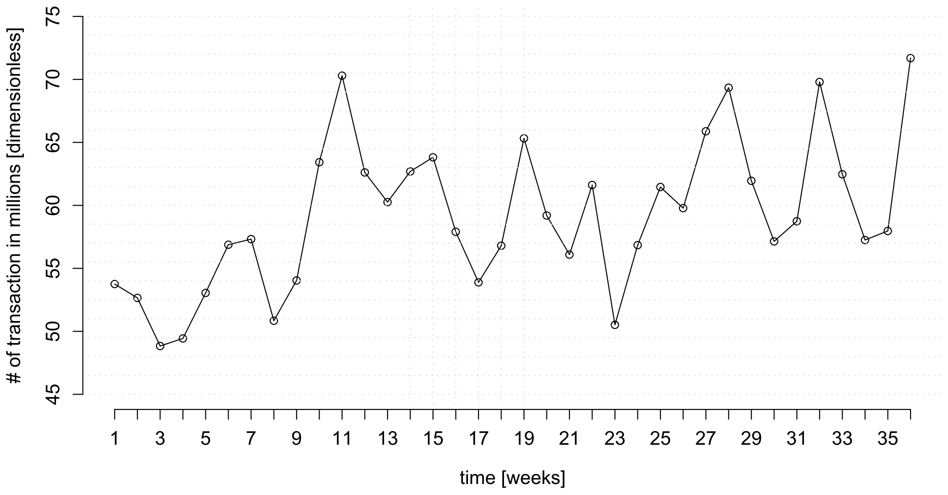

Fig. 6. Time series of the total volume of IMS transactions affected by the MIPS-reduction project.

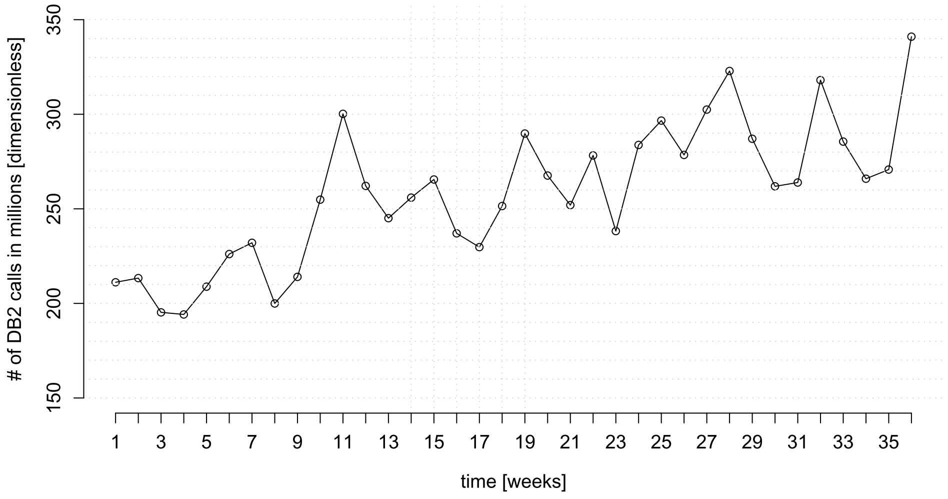

Fig. 7. Time series of the total volume of database calls generated by the IMS transactions affected by the MIPS-reduction project.

ings. As our analysis has shown in all of those modules the SQL constructs deemed as potentially inefficient were present. Furthermore, in all of the modules their presence was directly or indirectly linked with the improvements carried out.

Based on our analyses the observed reduction in the MSU consumption was linked with the MIPS-reduction project. The decrease did not happen due to a decline in either the number of database calls or transactions invocations. Also, other changes done to the code in the portfolio did not exhibit signs which would question the project's role in reducing MSU consumption. The observations derived from the MIPS-reduction project provide a real-world evidence that analysis of the code from the angle of the presence of potentially inefficient SQL constructs provides for discovering meaningful candidates for code optimization.

6. Portfolio-wide control

In this section we present how we approach control of the CPU resource usage at the level of a portfolio of mainframe applications. First, we show that for a large mainframe environment control of the CPU resource usage requires a structured approach. On the basis of the case study we show that both the proportions of the DB2 related code in the portfolio and the potentially inefficient SQL statements are high; hence making it clearly far from trivial to effectively carry out improvements. Next, we present how by combining the data available from the mainframe usage reports along with the source code extracted facts it is possible to narrow down the amount of sources that require scrutiny. For the portfolio under study we were able to restrict ourselves to less than 1% of all Cobol modules. To aid the process of locating the likely performance hampering Cobol modules we use the code derived data on the potentially inefficient SQL programming constructs and data concerning code interdependencies. Finally, we present how to assess the expected savings in MSU

Table 9
Potentially inefficient programming constructs found in all of the portfolio Cobol sources.

Construct ID	DB2 modules	% of DB2 modules	Instances	% of Instances
ORDER	1953	34.28	3232	23.72
$WHRH_2$	1327	23.29	2488	18.26
AGGF	945	16.58	2017	14.80
$WHRH_3$	744	13.06	1175	8.62
NWHR	704	12.36	1038	7.62
$WHRH_4$	379	6.65	573	4.20
$JOIN_2$	345	6.05	575	4.22
UNSEL	345	6.05	591	4.34
$WHRH_{5..10}$	345	6.05	637	4.67
COBF	276	4.84	369	2.71
DIST	160	2.81	266	1.95
GROUP	82	1.44	189	1.39
$JOIN_3$	79	1.39	141	1.03
UNION	41	0.72	54	0.40
$WHRH_{11..20}$	33	0.58	82	0.60
$JOIN_{4..7}$	27	0.47	53	0.39
$WHRE_3$	13	0.23	48	0.35
NIN	12	0.21	26	0.19
$WHRE_1$	11	0.19	22	0.16
$WHRH_{21..50}$	7	0.12	11	0.08
$WHRE_2$	6	0.11	19	0.14
$WHRE_{4..10}$	3	0.05	5	0.04
$WHRE_{11..30}$	2	0.04	8	0.06

consumption for particular code improvement projects. To enable quantification of the potential savings we constructed two formulas which are based on the data available from the evaluation of the MIPS-reduction project. These formulas serve as rules of thumb for estimating the percentage of the average monthly MSUs saved for a given IMS transaction given the number of altered DB2-Cobol modules. We illustrate how to plan code improvement projects by presenting two scenarios for source code improvements.

6.1. DB2 code

We analyzed the source code of the IT-portfolio to obtain insight into the state of the SQL code. The analysis covered 246 IT-systems which were built of 23,004 Cobol modules (the number does not include copybooks). Of these modules, DB2-Cobol programs constituted nearly 25% (5,698). The DB2-Cobol programs were identified by seeking the source code for the presence of the `EXEC SQL` blocks. In total we found 55,643 `EXEC SQL` blocks. Inside the blocks there were 14,387 `SELECT` statements which were used either as plain SQL queries or were embedded inside cursor declarations. The content of the remaining `EXEC SQL` blocks comprised SQL communication areas (SQLCA), `INCLUDE` statements, and the remaining SQL commands permitted in the Cobol programs' code.

We also measured the extent of the potentially inefficient programming constructs in the portfolio. In order to accomplish this task we applied our code analysis tools. Detailed results of this analysis are presented in Table 9.

In Table 9 we present results of the SQL code analysis which covered all the DB2-Cobol sources. In the first column we list the identifiers of the potentially inefficient SQL programming constructs. These identifiers are taken from Table 2. For the constructs: $JOIN_x$, $WHRE_x$, and $WHRH_x$ we created classes that cover ranges of instances. The ranges are determined by the minimum and maximum values for x. For instance, in case of the $WHRH_x$ one of the classes that we created is $WHRH_{5..10}$ which covers all instances of the $WHRH_x$ where x ranges from 5 until 10. By doing so we eliminated the large number of identifiers of constructs which occurrences were scarce; hence making the presentation of our table clearer. In the second column we provide the total numbers of Cobol modules in which the particular programming constructs were found. The following column, labeled *% of DB2 modules*, provides the percentage of the DB2 modules containing instances of the programming constructs with respect to all DB2 modules. In the column labeled *Instances* we give the total number of instances of the programming construct. And, in the last column we provide the percentage of the instances with respect to all instances of the potentially inefficient constructs. The counts and percentages relating to the instances are given only for information purpose. We did not utilize this information in our work. Rows of the table were sorted in a descending order according to the number of DB2-Cobol programs.

The primary observation that follows from Table 9 is that there is a large number of DB2-Cobol modules which contain the potentially expensive constructs in the embedded SQL code. We found approximately 68% (3874) of the DB2-Cobol programs with SQL statements in which at least one instance of the constructs listed in Table 2. In total we found 15,924 instances of such constructs. The top most occurring construct is ORDER. It is present in 1952 modules (34.26% of all DB2 programs). It tops the list both in terms of the total number of instances and modules it occupies. The high percentage of modules with the ORDER construct is somewhat alarming given that the corresponding SQL `ORDER BY` clause is known for increasing the probability of performing the CPU intensive operation such as sorting. From the MIPS-reduction project we know that presence of this construct did trigger improvement actions. Of course, the high incidence of ORDER constructs does not necessarily mean that the mainframe environment suffers from a large unnecessary MSU consumption. We did not conduct an analysis which would enable us to confirm or rule out such a possibility. However, this construct should be used with care.

Another construct which is worth discussion, for its relatively high position in the ranking, is the COBF. Let us recall that this construct refers to redundant calls made to the database. We have seen that due to the presence of this construct one module was altered during the MIPS-reduction project. Since use of the COBF is starkly redundant we found it surprising that 369 instances of this construct occur in 276 DB2-Cobol modules. Despite the fact that 276 constitute roughly 1.2% of the Cobol files it still appears as an overuse of a *bad* programming practice.

In the bottom of the Table 9 we see several variations of WHRE_x constructs. These constructs occur in a relatively small number of DB2-Cobol programs but due to their nature, which is nearly pathological, they deserve special attention. The positive information derivable from the SQL analysis is that we find only 24 modules which exhibit presence of WHRE_x constructs. Let us note that the given number is not the total of the numbers listed in the second column of Table 9 next to the WHRE_x identifiers. This derives from the fact that multiple variations of the construct may occur in a single module. Thus the actual reported total number of modules is lower than the total derivable from the table. Despite the low number of occurrences in the code these constructs are interesting from the perspective of CPU performance. In fact, in the MIPS-reduction project one module had a query with a WHRE_3 construct decomposed into three queries. These three queries were later embedded in the Cobol code in such a way that their invocations were controlled through a sequence of Cobol's `IF` statements.

From the analysis of the embedded SQL code we see a high incidence of potentially inefficient SQL programming constructs in the portfolio source code. The amount of modules, which on the grounds of presence of these constructs, speak for an in-depth analysis is huge. Such an amount constitutes a vast playground for SQL tuning experts. Of course, analyzing all the encountered cases and carrying out improvements, if necessary, is advisable from the software engineering perspective. The observations from the MIPS-reduction project show that presence of the constructs found in the portfolio poses a threat to efficient usage of the hardware resources. However, it is nearly infeasible for a portfolio of this size and such importance for the business to conduct such an overhaul of the source code. In fact, such a project also poses risks since code might become erroneous. Therefore, our advise is to carry out the impact analysis portfolio-wide, and mark the revealed code as such. Then if the code has to be changed anyway due to some business reason it is a good idea to combine this effort with code optimizations. By aggregating information about earlier code optimization efforts and their effects on the MSU consumption, this will become more and more routine and less risky. So we propose an evolutionary approach towards portfolio-wide control of CPU resource usage.

6.2. Major MIPS consumers

We now focus on a way to narrow down the search space for the modules with potentially inefficient SQL constructs. We propose to go after the low-hanging fruit and concentrate improvement efforts on the source code implementing the major MIPS consumers in the mainframe environment. Such an approach enables spending effort on those parts of the portfolio which provide for a relatively high savings potential. And, it also allows keeping improvements low risk for the business by limiting the extent of changes to the portfolio source code.

Accounting data In order to determine major MIPS consumers in a portfolio it is indispensable to have access to mainframe usage figures. Mainframe usage data, also known as the accounting data, are the essential input for fees calculation. The type of data which is required to conduct such calculations is collected by default. Apart from this essential data, z/OS also enables collection of all sorts of other data to provide detailed characteristics of particular aspects of the system or software products. Whether this is done or not depends on the demand for such data by the organization utilizing the mainframe. It is important to note that collection of the accounting data is a workload on its own, and also involves MSU consumption. This fact implies that careful selection must be made as to what is collected. For our analyses we had at our disposal weekly IMS transaction rankings of the top 100 most executed IMS transactions. This set of rankings was our primary source of data on the MSU consumption in the environment.

Transactions volume We propose to use the IMS transactions volume ranking to steer reduction of the MSU consumption in the IT-portfolio. The argumentation behind this decision is two-fold. Firstly, the general tendency is that the transactions volume is on the rise. This phenomenon finds its justification, for instance, in the fact that organizations gradually process more data using their IT infrastructure. Besides that clients of the organizations also get more used to the typically offered web front-ends to the IT-systems and as a result more data gets processed by the computers. In our case study the top used IT services have been experiencing increased CPU resource consumption over a long time frame. Management's attention has fallen mainly on five of those systems since more than 80% of MIPS were deemed to be used only there. Except for the increasing MIPS usage these systems are characterized by one other property. They are all interacting with the database. Nearly 57% of the transactions found in the top 100 IMS transactions were linked to those five systems. In a period of one year we have observed a 12.22% increase in the volume of transactions which involved these systems. We found no

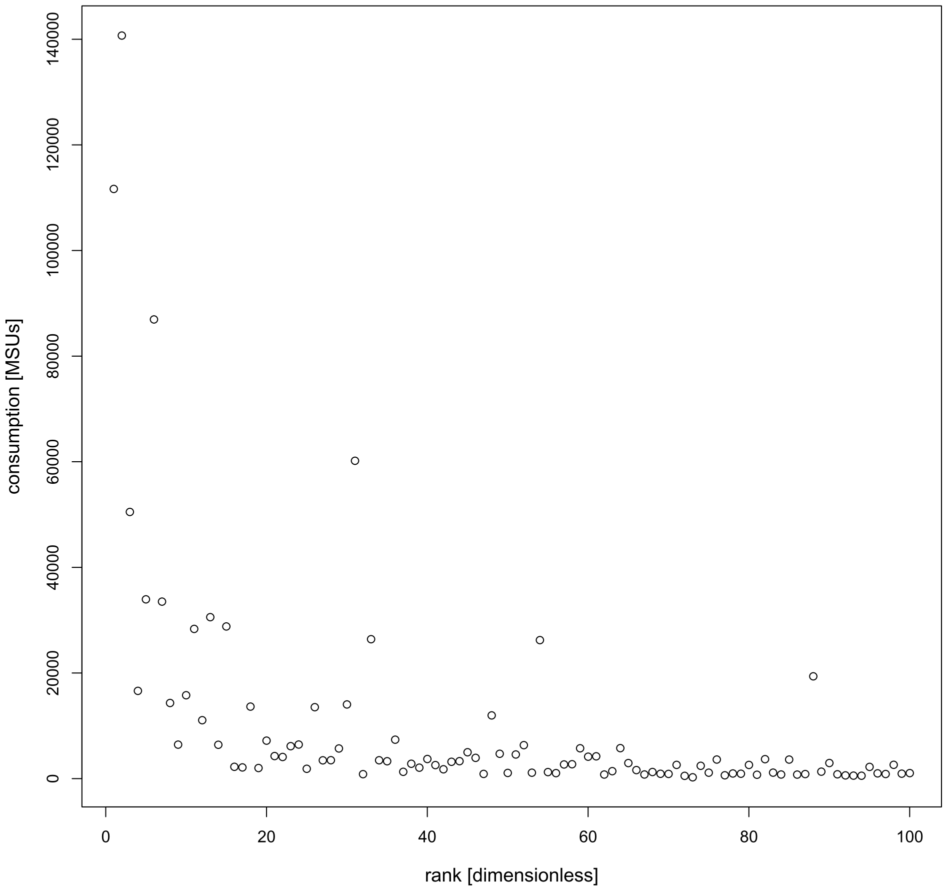

Fig. 8. Relationship between the transactions position in the ranking and the MSU consumption.

indication that this increase was temporary. In fact the systems which we analyze are the core systems of the organization. The increase of 12,22% most likely is a result of doing more business.

The other argument behind relying on the transactions volume to steer reduction of the MSU consumption is its relationship with the actual major MIPS consumers. We carried out an analysis of the MSU usage associated with the top most executed transactions. It turns out that the position of a top-ranking IMS transaction is a reliable indicator of the level of participation in the MIPS related costs. For this analysis we considered data reported in a period of six months for the top 100 most executed IMS transactions. For each transaction we calculated the totals for the number of transaction executions and the consumed MSUs in that period. We sorted the list of transactions in terms of the total number of executions in a descending order and plotted the associated MSU figures.

In Fig. 8 we present the observed relationship between the transactions position in the ranking and the associated total MSU usage in the period of six months. The horizontal axis is used to indicate the transactions position in the ranking. The values range from 1 until 100. The lower the value the higher the position in the ranking. A transaction with a higher transaction-volume obtains a higher ranking. The vertical axis is used to present the total number of MSUs consumed. From the plot it is visible that the observed MSU consumption tends to decrease along with the position in the ranking. Although we see several exceptions, this observation suggests that the transaction's frequency of execution is key in the transaction's contribution to the MIPS related costs.

The fact that MSU usage is positively related to the volume of executions is not surprising. Execution of each transaction is associated with some consumption of MSUs. The amount of consumed MSUs varies from transaction to transaction. Varied usage derives from the fact that CPU utilization depends on the actual number of instructions passed to the processor. This is ultimately determined by the logic embodied in the code of the executed programs implementing the transaction and data flow. The peaks in the ranking which are noticeable for several transactions in the plot in Fig. 8 are due to the varied MSU consumption. Nevertheless, we assume the ranking as a sufficiently good tool to locate major MIPS consumers in the IMS environment and rely on it in the process of pinpointing source code for the DB2 related improvements.

Based on our observations, from this and other case studies, a software portfolio typically contains only few applications which share some common property, e.g. play a major role in supporting business. This phenomena follows the well-known 80-20 rule, or Pareto principle, which roughly translates into: 80% of the effects come from 20% of the causes [44]. Whereas

Table 10
Potentially inefficient programming constructs found in the Cobol sources which implement the top 100 IMS transactions.

Construct ID	DB2 modules	% of DB2 modules	Instances	% of Instances
ORDER	72	1.26	134	0.98
$WHRH_2$	71	1.25	155	1.14
$WHRH_3$	50	0.88	84	0.62
AGGF	43	0.75	136	1.00
$WHRH_4$	24	0.42	36	0.26
NWHR	22	0.39	44	0.32
$JOIN_2$	21	0.37	44	0.32
$WHRH_{5..10}$	15	0.26	34	0.25
COBF	14	0.25	17	0.12
UNSEL	13	0.23	22	0.16
DIST	10	0.18	33	0.24
$WHRE_3$	9	0.16	34	0.25
$JOIN_3$	8	0.14	26	0.19
$WHRE_1$	8	0.14	18	0.13
GROUP	5	0.09	7	0.05
$JOIN_{4..max}$	5	0.09	17	0.12
UNION	4	0.07	5	0.04
$WHRE_{4..10}$	2	0.04	2	0.01
$WHRE_2$	1	0.02	1	0.01
$WHRE_{11..max}$	1	0.02	6	0.04
$WHRH_{11..20}$	1	0.02	6	0.04

there is no hard proof to support this claim this observation appears to hold true for most portfolios. For the case study used in this paper we found that the top 100 IMS transactions contain nearly 57% of transactions linked to IT-systems which are attributed to more than 80% of MIPS usage. Whereas the situation encountered is not exactly in line with the Pareto principle, application of the principle to the investigated problem is straightforward.

DB2 transaction Among all IMS transactions that we find in the top 100 ranking we concentrate, for obvious reasons, on those which interact with the database. We will refer to an IMS transaction as a DB2 transaction each time we find in its implementation code which is related to DB2. Given that we approach the IT-portfolio from the source code perspective to determine which IMS transactions are DB2 transactions we screen the content of source files. Namely, we first map the IMS transaction with source files that implement its functionality. Next, we scan the files and seek for the presence of particular programming constructs which are meant to implement interaction with the database. A typical DB2-client program written in Cobol contains embedded SQL code which is enclosed in `EXEC SQL` blocks. We use this fact to classify Cobol programs as DB2-clients. Finally, once at least one source module containing DB2 related code is found in the implementation of an IMS transaction it is marked as DB2 transaction.

Given that the set of DB2 transactions was chosen on the basis of static code analysis we allow in the set those IMS transactions which in runtime never establish any connection with the DB2. This situation derives from the fact that logic in the implementation of the transactions ultimately dictates when a DB2 related code is executed. This lack of accuracy is, however, unavoidable given the kind of analysis we deploy in our approach. Nevertheless, by distinguishing between DB2 and non-DB2 related IMS transactions we restrict the number of source modules to those that are relevant for further analysis.

6.3. *Low-hanging fruit*

Conducting DB2 related improvements inevitably boils down to reaching for the relevant part of the source code. So far we have explained that for the purpose of focusing improvement efforts we restrict ourselves to the top 100 IMS transactions which we dubbed the major MIPS consumers. Having at our disposal the set of Cobol modules which implements the transactions we carried out analysis of the embedded DB2 code. We now present results of this analysis. As it turns out the code which is relevant from the perspective of improvements constitutes, in fact, a small portion in the portfolio.

Top executed DB2 code In an earlier part of this paper we presented results of the analysis of the SQL code which was spanned across all the Cobol modules found in the portfolio. In the context of this section we restrict our analysis only to those DB2 Cobol modules which are associated with the implementation of the top 100 IMS transactions. Let us recall, in total the implementation comprises 768 Cobol modules. Of these modules 258 contain DB2 related code.

In Table 10 we present results of the SQL code analysis from those DB2-Cobol sources which implement the top 100 IMS transactions. In the first column we list the identifiers of the potentially expensive SQL programming constructs. These identifiers are taken from Table 2. Again, for the constructs: $JOIN_x$, $WHRE_x$, and $WHRH_x$ we created classes that cover ranges of instances. In the second column we provide the total numbers of Cobol modules in which the particular programming constructs were found. The following column, labeled *% of DB2M*, provides the percentage of the DB2 modules containing

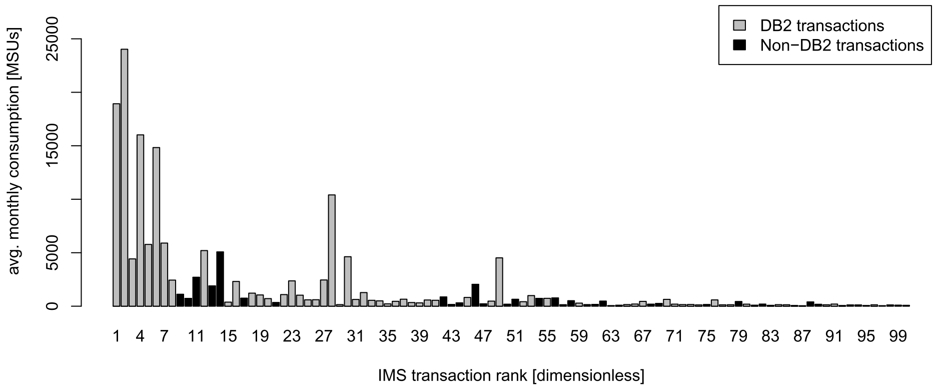

Fig. 9. Bar plot of the monthly MSU consumption for each IMS transaction in the top 100 ranking.

instances of the programming constructs with respect to all DB2 modules. In the column labeled *inst* we give the total number of instances of the programming construct. And, in the last column we provide the percentage of the instances with respect to all instances of the potentially inefficient constructs. Rows of the table were sorted according to the number of DB2-Cobol programs.

In Table 9 the modules which contain the ORDER construct constitute over 34% of all DB2-Cobol modules whereas in Table 10 we see roughly over 1%. Table 10 exhibits similarity with Table 9 in terms of order of the programming constructs. Again, the ORDER construct tops the list, the variations of the WHRH$_x$ constructs occupy its upper part, the COBF resides around the middle. Also, majority of the positions in the bottom of the list is, again, occupied by the WHRE$_x$ constructs. Of course, in Table 10 the proportions of the modules and instances are much lower but this is due to the fact that we look at a much smaller portion of the portfolio. What is particularly interesting are the modules labeled with WHRE$_x$. In the entire DB2-Cobol part of the sources we found 24 modules which contain instances of those constructs. This time we screened 258 files and we see 21 modules with this property. Given that the WHRE$_x$ constructs are pathological it is surprising to learn that 87.5% of source files implementing the top IMS transactions contain such code. Despite the presence of nearly a full array of SQL constructs deemed as potentially inefficient the overall number of DB2 modules with such constructs is rather low (180). It is fair to say that in the worst case scenario code improvements are restricted to at most 180 Cobol programs. Given the total number of Cobol modules in the analyzed portfolio (23,004) this constitutes approximately 0.78%.

The small portion Analysis of the mainframe applications' SQL code revealed that there are plenty of opportunities to fine-tune interaction with the database. As we have already explained when dealing with a business critical portfolio a preferred approach is to focus on achieving savings at low-risk. In general, the lower the amount of changes applied to the code the lower the risk of encountering defects in the applications. In case of changes to the DB2-related code, alterations are normally restricted only to a few lines of the embedded SQL code. These minor changes can be sufficiently tested without too much effort prior to migrating modules to the production environment; hence making them relatively low-risk alteration of the applications.

In Fig. 9 we present a plot of the monthly MSU consumption for each transaction in the top 100 ranking. The horizontal axis is used to provide the rank of an IMS transaction according to monthly transactions volume. The data presented in the plot were captured in a particular month. The vertical axis is used to present the observed consumption of MSUs by the transactions. In the bar plot we distinguish two types of transactions: those which interact with the database (gray) and those which do not (black). In the studied pool of IMS transactions 64% interacts with the database. Similarly as in Fig. 8 we see that the high-ranking IMS transactions tend to have higher MSU consumption. This is not a one-to-one correspondence. We see some deviations. What is interesting in this plot is that it is visible that the DB2 interacting IMS transactions have a distinctly higher MSU consumption than other IMS transactions.

We took the top 100 IMS transactions listed in Fig. 9 and analyzed them from the perspective of the amount of source modules present in their implementation. In the first step we mapped with each IMS transaction a set of modules used for their implementation. Each set was associated with a position number identical to the rank the corresponding IMS transaction occupied in the ranking. Then, for each position i we computed the cumulative number of distinct modules which exist in the implementations of transactions ranked 1 through i. This operation was accomplished by calculating the union of the corresponding sets. In addition, for each position we obtained cumulative counts of the DB2-Cobol modules and the DB2-modules with the potentially inefficient programming constructs. Eventually, we had at our disposal three vectors of data containing the cumulative counts.

In Fig. 10 we present the results of our analysis of the code implementing the top 100 used IMS transactions. The horizontal axis lists the IMS transaction rank in terms of execution volume and the vertical axis provides the Cobol modules cumulative counts. The counts of the modules implementing the transactions reflect: the total number of Cobol modules (solid line), the DB2 interacting modules (dashed line), and the DB2 interacting modules containing the potentially inefficient SQL code (dotted line).

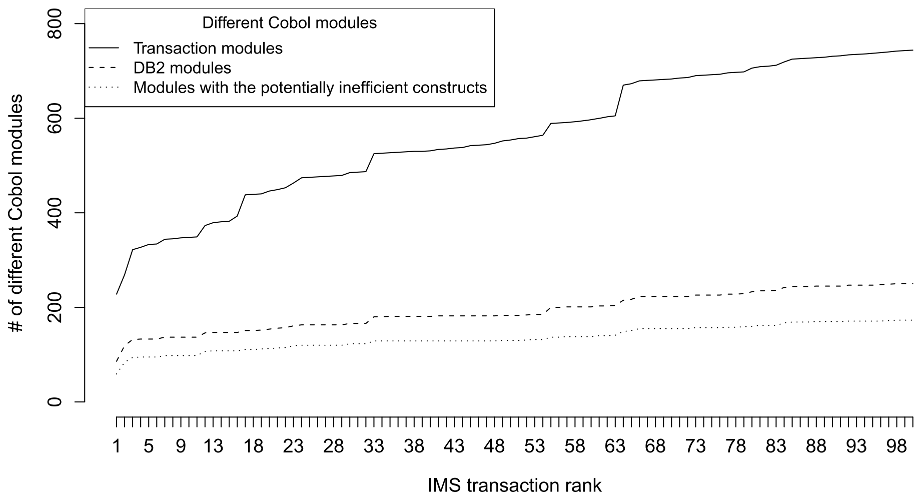

Fig. 10. Properties of the Cobol code implementing the top 100 IMS transactions.

What is observable from the plot is that a relatively large number of the major IMS transactions are implemented by a rather small number of Cobol programs (768). In the global portfolio view these modules represent approximately 3.2%. An even smaller number of code is responsible for interaction with the DB2 (258 modules, 1.07%). This follows from the fact that there exist many code interdependencies among the implementations of the IMS transactions. Finally, we see that among the DB2 modules implementing the top 100 IMS transactions, 180 (0.78%) contain the potentially inefficient SQL constructs. Bearing in mind the plot in Fig. 9 we see that in order to address the top MSUs consuming DB2-transactions it is sufficient to take into consideration a relatively small portion of the code. By taking into account different subsets of the IMS transactions it is possible to arrive at various configurations of Cobol modules which are worth inspection. These configurations will possibly differ in the numbers of modules, and what follows from it, the risk associated with optimizing them and migrating to the production environment. We will now show how we utilize the code extracted data in planning a MIPS-reduction project.

6.4. Improvement scenarios

We use the data gathered during the analysis of the code and show how to use it to plan source code improvement projects. We illustrate portfolio code improvements on the basis of two what-if scenarios. For each scenario we present estimations of the possible savings.

Estimation tools In order to be able to estimate the potential savings in MSU consumption resulting from carrying out a code improvement project we need some benchmark data. The only data that were at our disposal on that kind of projects came from the MIPS-reduction project evaluation report. We chose to rely on it in estimating the potential MSU savings for the similar future projects. Our analyses so far have been revolving around optimization of the MSU consumption by the IMS transactions through improvements to their implementation. Using the available data we drafted two formulas for estimating the percentage of the monthly MSUs saved for executing an IMS transaction given the number of modules that were altered in its implementation. One formula provided us with the average expected monthly percentage of savings, the other, with the maximum.

We considered savings calculations provided in the evaluation of the MIPS-reduction project by the expert team. We had at our disposal percentages of the MSUs saved and the average monthly numbers of MSUs consumed by the transactions. These were obtained from the historical yearly data. In order to associate this data with the code alterations, for each transaction we counted the number of modules changed in the implementation of the transaction during the MIPS-reduction project. The counts ranged from 0 up to 4 (0 applied to the one exception we reported earlier). We grouped the transactions according to the number of modules altered. We eliminated groups which contained less than one data point. Eventually, we arrived at 3 groups of transactions with 1, 2, or 3 modules improved. We characterized each group with two values: the average and the maximum percentage of MSUs reduced. For each group the average was calculated as the weighted average of the reported percentages with the historical average monthly MSU consumption as weights.

We chose to do extrapolation on these small sets of three points. Although, the linear relationship appears to be appropriate given the points we had we did not find it suitable. Whereas we expect the increase in MSU reduction as we improve more and more Cobol modules we do not expect this growth to be proportional. We decided to choose logarithms as the most appropriate family of functions to extrapolate. Logarithmic functions are increasing, if the base is larger than one, and their growth is very slow. This property stays in line with our intuition. Having assumed nonlinear behavior we used the available data points to extrapolate. We found the following relations.

Table 11
Summary of the analyses of the scenarios.

Scenario	Modules (#)	Est. monthly savings (%)	
		Avg	Max
Major overhaul	180	9.61%	16.83%
Minor changes	14	6.07%	9.56%
MIPS-reduction project	6	*3.84%*	–

$$R_{avg}(m) = 0.010558 \cdot \ln\big(3561.230982 \cdot (m+1)\big). \tag{1}$$

$$R_{max}(m) = 0.0512401577381 \cdot \ln\big(4.138711 \cdot (m+1)\big). \tag{2}$$

Formulas 1 and 2 provide the tools we used for estimating the expected MSU reduction in altered transactions. The R_{avg} stands for the average expected percentage of reduction. And, the R_{max} gives us the maximum expected. Both formulas use as input m which is the number of Cobol modules changed in a transaction's implementation.

In our approach to estimating savings we rely on the fact that source code is altered. We size the extent of code alterations with the number of modules changed. We do not take into account the actual scope of changes, for instance, the number of SQL statements improved within a particular module. This is obviously a simplification. As an alternative one might consider counting the number of changed lines of code. However, such approach bears its own drawbacks and given the small number of data points does not guarantee achieving any better estimates. In our case by relying on limited data we were able to equip ourselves with tools which enable us to make estimations based on the past experience. Let us note that the formulas were derived from characteristics of a specific production environment and therefore it is possible that for other environments one must consider devising their own using different data. Nevertheless, in the face of having no approximation tools we came up with formulas that we use as rules of thumb for estimating the size of savings obtainable from DB2-related code improvements.

What-if scenarios Naturally, there exist various scenarios in which it is possible to achieve reduction in MSU consumption through code improvements. Each scenario is characterized with a level of potential savings and a scope with which the applications' source code is affected. The first element has obvious financial consequences. The latter relates to IT risk. Two scenarios for code improvement in the implementations of the major IMS transactions were analyzed. To estimate savings we used the formulas (1) and (2). For the sake of analysis we assumed that the basic improved code unit is a DB2 Cobol module. The costs relating to code improvements are assumed to be low compared to potential savings. The following are the analyzed scenarios:

Major overhaul We focus on the top 100 IMS transactions. All DB2 modules bearing the potentially inefficient SQL constructs found in code analysis are improved. The emphasis in this scenario is on code quality improvement. Savings in MSU consumption are expected to follow as a consequence of improvements to the code.

Minor changes The number of DB2 modules dispatched for improvements is kept low yet the impact of the alterations made to these modules allows embracing IMS transactions which offer potential for high savings. Selection of modules for this scenario was based on facts such as presence of potentially inefficient SQL constructs and code interdependencies among implementations of the IMS transactions. The emphasis in this scenario is on achieving significant savings in MSU consumption through relatively small number of alterations in the applications' code.

In Table 11 we present a summary of the scenarios from the perspective of the total potential savings and the amount of source code which is affected. In the first column we list the scenarios. In the second we provide the number of Cobol source modules which undergo improvements. In columns 3 and 4 we present the estimated savings. The savings are expressed as the percentage of the average monthly MSU consumption by the top 100 IMS transactions. The scenarios are summarized on the basis of calculations obtained from formulas 1 and 2. The columns contain labels *Avg* and *Max* to distinguish figures obtained with the two formulas. In the last row of Table 11 a summary of the MIPS-reduction project is provided. The figures given are the actuals and therefore cells in columns labeled with *Max* do not contain any value.

It is clear from Table 11 that in all scenarios MSU consumption savings are to be expected. In principle, the more changes to the code the higher the projected yields. The highest expected savings are generated in the *Major overhaul* scenario. These savings come for the price of in-depth analysis and alterations to a large number of modules. An interesting result is revealed through the *Minor changes* scenario in which only 14 modules are considered yet the expected savings are substantial. In the last row estimates regarding changes to the same Cobol modules that were changed in the MIPS-reduction project are presented. Let us note that the value is expressed as the percentage of the average monthly MSU consumption by all the top 100 IMS transactions.

Which scenario is the best to depart from depends on a number of factors. For instance, on the desired level of savings, the risk tolerance, or the degree to which the source code is to be improved. Nevertheless, the analyses presented here allow for rational decision making based on facts.

The fact that the savings in the MIPS-reduction project were lower than those we estimated for the two analyzed scenarios is primarily due to differences in the scope of the portfolio source code involved. Our approach is unlike the expert team. We departed from screening the code which implements the top 100 IMS transactions and marked the potentially inefficient code. The MIPS-reduction project team was constrained by the business domain which commissioned the project. What follows from it is that they focused on a particular subset of the IMS transactions. From there they made the selection of the code to be improved. Taking the IT-portfolio management point of view, we would recommend not to constrain code improvements by the business views, such as the business domains. In a mainframe portfolio of applications serving multiple business domains the source code constitutes a uniform space for searching improvement opportunities. The mainframe usage reports are already a reliable guide which points to the major cost components. There is no need, from the technology perspective, to impose artificial constraints when the goal is to lower the operational IT costs.

7. Practical issues

In order to control CPU resource consumption and generate savings, the control efforts must be embedded within the organization and in the ongoing software process. We already alluded to that by proposing an evolutionary approach towards MSU consumption reductions: monitoring, marking and changing in combination with other work on the code. Since there will be new modules and enhancements, there will also be new venues for reductions in CPU resource usage, and we find those during monitoring. It would be good to analyze new code upfront for this to preventively solve the problem before it occurs. Of course, in order to take advantage of all those opportunities there must exist a clearly defined software process.

Within the software process two phases should be distinguished. The first phase involves code improvements to the existing implementations of the MIPS consumers. The second phase involves ongoing monitoring of changes in the production environment MSU consumption, and also, monitoring of the quality of the code in the pre-production stage.

7.1. Code improvements

As the MIPS-reduction project has shown, code improvements yield significant savings. The MIPS-reduction project was restricted to a particular group of IMS transactions. We have demonstrated that by carrying out code improvements on a portfolio scale it is possible to achieve higher reduction in the MIPS related costs. These facts speak for themselves in managerial terms. In addition, what follows from the code improvements is also the increase in the quality of the source code. For an organization this aspect translates, for instance, into prolonged longevity of the portfolio, better response time of the applications, or better control of the IT-assets. In order to commence with the CPU resource usage control the current condition of IT must undergo screening.

Implementation of this phase could take form of one of the source code improvement scenarios analyzed in this paper. Of course, for a particular mainframe environment there needs to be data available which would allow regeneration of the results used to construct the improvement scenarios presented in here. Implementation of a portfolio-wide code improvement project should take into account lessons learned from analysis of the entire Cobol written portfolio, the MIPS-reduction project, or recommendations from the portfolio experts.

To facilitate analysis of the Cobol portfolio we developed a number of tools which allow for extraction of the relevant data from the source code. These tools are capable of delivering an analysis each time changes to the portfolio take place. In fact, these tools are suited for analysis of many Cobol sources. Minor modifications might be necessary. Of course, except for the collection of the code related data it is also necessary to have at your disposal the mainframe usage data. This data, however, is usually available for all mainframe environments. If not their collection needs to be enabled.

7.2. Ongoing monitoring

There is a clear need to monitor CPU resource usage on an ongoing basis. It has been observed that source code in IT-portfolios evolves. According to [51] each year Fortune 100 companies add 10% of code through enhancements and updates. In the Cobol environments alone, which are estimated to process approximately 75% of all production transactions, the growth is also omnipresent [4]. In 2005 experts at Ovum, an IT advisory company, estimated that there were over 200 billion lines of Cobol deployed in production [5]. They claimed this number was to continue to grow by between 3% and 5% a year. There is no sign that the Cobol's growth has stopped. Our own study revealed that in the production environment we investigated in a single year the number of Cobol modules grew by 2.9%.

Apart from changes to code there are also changes relating to the systems usage patterns. For instance, the MIPS capacity attributed to the one system in the studied portfolio has increased nearly 5 times in just 3 years time. For this particular system it was observed in time that many IMS transactions, which were linked to the system, entered the ranking of the top 100 executed IMS transactions.

The following are major elements involved in the ongoing monitoring of CPU resource usage with respect to DB2 linked IMS transactions:

- Enforcement of the verification of the SQL code in the testing phase to eliminate the inefficient code from entering the production environment.
- Utilization of the mainframe usage data for identification of changes in the IMS transactions usage patterns.

– Deployment of a mechanism which enables measuring MSU consumption or CPU time spent on execution of particular SQL statements by the IMS transactions.

There are multiple possibilities to realize those elements. For instance, the verification of the SQL code prior to its entry to the production environment can be accomplished by conducting checks of the embedded SQL code as it has been proposed in this paper. Our list of potentially inefficient programming constructs was derived based on the DB2 experts recommendations and the real-world examples taken from the MIPS-reduction project. The content of this list has been likely not exhausted. In fact, we would recommend an evolutionary approach here. Each time any kind of project which involves alterations to the mainframe applications code is carried out some increment to the knowledge on the inefficient code constructs could be made. For particular mainframe environments the insights based on organization's internal experience provide the most accurate information on the actual bottlenecks in the source code.

To enable analysis of the IMS transactions usage patterns it is important to assure that relevant mainframe usage data are collected. This is typically done by enabling adequate mechanisms on the z/OS. In case of the organization which provided us with the case study the collection of the accounting data was well established. Partly due to the regulatory framework in which the business operates. Nevertheless, having the data available is not sufficient to take advantage of it. There must be clearly defined paths which allow exploitation of its abundance. For instance, ongoing measurement of the average MSU consumption by the transactions in the IMS environment would set up a foundation for enabling automated reporting of the deviations from the historical means.

Being able to capture the performance of execution of particular statements is somewhat more involving. It boils down to deployment of monitoring techniques which rely on the feedback from the operating system. z/OS provides facilities which enable capturing all sorts of detailed data bits through the so-called IFCID (Instrumentation Facility Counter ID) record blocks. For instance, IFCID 316 captures metrics with respect to SQL statements executed in DB2. Among these metrics we find information on the CPU time [38]. By capturing and storing the IFCID 316 data it is possible to deploy a mechanism to measure CPU usage by the particular SQL statements. Later on it is possible to use this data for performance analysis, and eventually planning improvements. Of course, all sorts of monitoring deployments must be carefully arranged with the mainframe experts to make sure that such mechanisms alone will not result in undesired CPU resource leaks.

8. Discussion

In this section we consider the source code based control of CPU resource usage in a broader spectrum of IT-management issues. First, we position our approach in the context of vendor management. In particular, we discuss how this domain of IT-management can benefit from the work presented in this paper. Second, we discuss mainframe utilization from the perspective of potential redundancies. We focus on the usage of various environments within mainframes and discuss how to address the potential inefficiencies.

8.1. Vendor management

Trust is the foundation of any business venture. However, when the stakes are high additional means must be considered to assure success. For business, which is critically dependent on IT, software assets are priceless. Therefore, organizations which outsource IT activities to external parties must keep track of what happens. Lax or no control over outsourced IT is likely to lead to loss of transparency over its condition. It is possible to base evaluation of IT deliverables on data provided by the outsourcing vendor. However, such data is at clear risk of being distorted and not present a true picture. The bottom line is that it is in the best interest of the business management and shareholders to possess reliable means to control the state of IT.

Source code represents the nuts and bolts of the software assets. Maintaining its high quality is paramount since it has impact on numerous IT aspects. For instance, in our work we enabled the reduction of IT operational costs through improvements in the source code. We relied on the fact that inefficient code, or in other words, low quality code hampers the usage of hardware resource. Outsourcing IT frequently means outsourcing the alterations done to the source code. Of course, no CIO is expecting to degrade quality of the code by outsourcing it. In order to prevent from such distortions to happen organizations must equip themselves with adequate quality assurance mechanisms.

Source code quality assurance at the level of vendor management typically boils down to two aspects: provisions in the service level agreements (SLAs) and methods for compliance verification. Provisions of an outsourcing SLA must stipulate in what condition the source code is expected to be delivered by the vendor. In order to make these conditions verifiable they better be expressed in source code terms. What we mean by that is a clear outline of the kind of code metrics, set of statements, or language dialects, to name a few, the delivered code should adhere to. This way it is possible to devise means to verify that the conditions of the SLA have been met.

Verification process can take two routes. The organization outsourcing its IT relies on the compliance assurances made by the vendor, or it chooses to verify the compliance on its own. We would recommend the latter. Having the source code quality conditions expressed in the source code terms makes it possible to deploy source code analysis techniques for verification. Such approach provides for the highest level of transparency on the source code. If the source code quality

Table 12
Breakdown of the MSUs usage of the top 100 transactions into 4 IMS environments.

Month	PROD	Environment		
		TEST1	TEST2	TEST3
1	87.90%	10.82%	0.43%	0.85%
2	84.34%	14.52%	0.63%	0.51%
3	92.26%	6.02%	0.97%	0.74%
4	87.82%	10.62%	0.65%	0.90%
5	91.86%	6.35%	0.66%	1.13%
6	95.44%	3.03%	0.73%	0.80%
Monthly average	89.94%	8.56%	0.68%	0.82%

criteria are specified in such a way that lexical analysis of the code is sufficient than verification process is implementable in an inexpensive manner.

In our approach to reducing CPU resource usage we used the lexical code analysis to mark the potentially inefficient code in the Cobol files. In our case we focused only on the use of SQL. Of course, it is possible to check other aspects of the code following this very approach. The tooling we used for our purposes can be naturally deployed to verify SQL code in the process of controlling the outsourcing vendors.

A real-world example In the portfolio we study in this paper we encountered a number of DB2-Cobol modules with instances of the WHRE_x construct. We inspected all the modules which contain this construct in order to learn about the origins of the construct. We looked for the common properties these files shared. Since the number of modules was small it was feasible to analyze them manually. First, we analyzed the queries labeled with WHRE_x. As it turned out two thirds of the modules containing the queries originated from the implementation of the same system. We checked the queries for duplicates, which would suggest a *copy-paste* approach to programming, but out of 102 instances of the WHRE_x we found only 1 case of a verbatim copy. This simple experiment suggests that the undesired WHRE_x constructs did not propagate into the code through code duplications. Second, we scrutinized comments embedded in the 24 modules to track the history of changes. For 21 modules we found dates and identifiers of the programmers who were indicated as authors of those modules. We found 9 different authors among whom one was associated with as many as 13 modules. As it turned out the name of the author pointed to a company to which projects involving source code modifications were outsourced. It appeared as if the programmer (or programmers), who labeled these 13 modules with the name of the company, smuggled this coding malpractice into the portfolio. Presence of the WHRE_x instances in the code and the comments embedded version history details expose interesting information. Of course, they neither prove that the WHRE_x constructs were in the modules since their inception into the portfolio nor guarantee that the outsourcing company put them there. However, these observations provide basis to raise questions as to the quality of work delivered by the vendor.

8.2. CPU resource leaks

Large organizations use mainframes to accommodate business processes with heavy workloads. There is a whole array of tasks that a typical mainframe accomplishes. Hosting of data warehouses to facilitate data mining, offering web-enabled services, processing batch-jobs, handling IMS transactions, storing data, to name a few. Each such task contributes in some degree to the overall mainframe usage bill. Within the production environment we distinguish two types of workloads: business critical and non-business critical jobs. The first type relates to applications which require the high-performance and high-availability provided by a mainframe. The second type refers to all other jobs which are not critical for the business or do not necessarily require the reliability a mainframe provides. The second type refers to all other tasks which are executed on the mainframe but are not critical for the business processes or not necessarily require the top-notch execution performance. The second type of tasks constitutes the area with potential redundancies in MSU consumption. In other words, an area where CPU resources leak. Identification of such areas yields a way to further reduce mainframe utilization, and what follows, the related costs.

In this paper we concentrated our analysis on the source code of IMS transactions executed in the production environment. All of them were highly critical for the business and without any doubt they had to be executed on the mainframe. On the mainframe the organization also developed and tested software which later ended up in production. Let us now illustrate the distribution of the MSU consumption among the environments designated for IMS. The MSUs were reported for four IMS transaction environments: PRODUCTION, TEST1, TEST2, and TEST3.

Table 12 shows the breakdown of the MSUs usage of the top 100 transactions into 4 IMS environments. In the first column we indicated the month number. In the remaining columns we provide percentages of the total monthly MSUs consumed in each IMS environment. In the last row the arithmetic average for each listed environment is provided to indicate an average monthly percentage of MSU consumption in the entire period.

Not surprisingly, the largest portion of the MSU consumption is associated with the production environment. On the average it accounts for nearly 90% of all MSUs consumed. All the remaining environments consume roughly 10% with the `TEST1` environment consuming the majority of approximately 8%. We implicitly assumed that the workloads on the production environment are critical and cannot be removed. The workloads assigned to the test environments do not serve business processes in a direct way yet they take up approximately 10% of the IMS environments MSUs. These MSUs loom as an easy to eliminate target. An obvious approach would involve moving the test workloads onto environments which do not incur MSU related charges, such as PCs. It has been a reality in many IT shops since shortly after the invention of the PC to migrate from MSU consuming environments. Compilers, language parsers, test data generators, testing environments and source code libraries have all been available on PCs and servers for years. Many have syntax specifically geared to the IBM mainframe environment. In case of this portfolio migration of testing was complicated. So that this solution was not possible for this organization to implement. Nonetheless, insight like the one presented here certainly provides food for thought for mainframe cost reduction strategists.

9. Conclusions

Costs relating to mainframe usage are substantial. Despite this fact the CPU resource consumption is not managed in a granular fashion. We departed from the fact that the usage charges are directly dependent on the applications' code and leveraged the code level aspects up to the executive level of operational costs. In this paper, we presented an approach to managing MIPS related costs in organizations which run their applications on mainframes.

Our approach relies on source code improvements relating to interaction between the applications and the database. Its distinct feature is that it does not require instrumentation of the mainframe running the applications, what allows eliminating risks that can jeopardize continuity of operations. Also, it allows obtaining the insight into the mainframe environment and conduct planning of code optimization projects without the actual need to access the mainframe. One of our assumptions was to be pragmatic so that facilitation of our approach in an industrial setting is feasible. We achieved it by relying on the type of data that a typical mainframe operating organization possesses: source code and the mainframe usage information. We showed that our approach is adequate to incorporate management of CPU resource usage at the portfolio level.

We investigated a production environment running 246 Cobol applications consisting of 23,004 Cobol programs and totaling to 19.7 million of physical lines of code. Approximately 25% of all the Cobol programs interacted with DB2. Our investigations were focused on the IMS-DB2 production environment. In particular, we studied the MSU consumption figures relating to the top 100 most executed IMS transactions. One of the characteristics of the IMS environment was that on a weekly basis between 72%–80% of all the major IMS transaction invocations involved database usage. As it turned out the MSU consumption for those transactions was on average higher by more than a half compared to the consumption reported for the non-database related invocations.

An earlier small scale MIPS-reduction project triggered our full-scale portfolio analysis. As the project showed, through SQL tunning it was possible to save approximately 9.8% of the annual cost linked to executing the optimized portion of the portfolio. Our approach enabled us to effectively constrain the search space for inefficient SQL code in a large set of source files. To achieve this we bridged the source code dimension with the financial dimension. We related the mainframe usage data to the source code implementing the IMS transactions.

The combination of SQL programing knowledge, findings from the MIPS-reduction project, and input from the experts gave us a set of syntactic rules which enable filtering out the potentially inefficient SQL constructs from the portfolio sources. We showed how to use those rules along with the code interdependencies information to narrow down the number of source files potentially worth optimization. With our approach we could identify as little as 0.78% of all modules as candidates for actual improvements. We presented our tooling in detail so that others can use our approach in their own context. As we showed the tooling is simple to implement.

We demonstrated two code improvement scenarios and calculated the potential reductions in MSU consumption. We showed that by selecting as little as 14 Cobol-DB2 modules there exists a possibility of cutting approximately 6.1% of the average monthly MSU consumption in the studied environment. By carrying out a more extensive code improvement project involving a potential 180 modules the savings can reach as much as 16.8%.

Our work presented here provides organizations with the following. First, it outlines a framework for incorporation of a structured approach to CPU resource management on the mainframe. Second, it provides for improvement of code quality through enforcement of usage of the SQL coding guidelines derived from a substantial real-world portfolio. Finally, it demonstrates how to use information obtained from source code analysis and mainframe usage data to plan projects aimed at reducing MSU consumption.

To conclude, identifying the few most promising Cobol modules that give opportunity to significantly reduce CPU resource consumption is a viable option for reducing operational IT costs. We provided an approach, tooling and an example to implement this. Our approach leads to fact-based CPU resource management. We hope that it will become a trigger for the IT executives to increase operational IT efficiency.

Acknowledgements

This research received partial support by the Dutch *Joint Academic and Commercial Quality Research & Development (Jacquard)* program on Software Engineering Research via contract 638.004.405 *EQUITY: Exploring Quantifiable Information Technology Yields* and via contract 638.003.611 *Symbiosis: Synergy of managing business-IT-alignment, IT-sourcing and off-shoring* *success in society*. We would like to thank the organization that will remain anonymous for kindly sharing their source code and mainframe usage reports with us. We are also indebted to the employees of the IT department for providing us with their support. Patrick Bruinsma, from IBM Netherlands, for providing us with comments and expertise with regards to mainframe environments. Last but not least, we are very grateful to our colleague Rob Peters for meticulously reviewing this paper numerous times.

References

[1] MACRO 4, MACRO 4: Users 'slam' vendors for not helping to control mainframe processor consumption. TradingMarkets.com (January 2009) Available via http://www.tradingmarkets.com/.site/news/Stock%20News/2122225/.
[2] Peter H. Aiken, Reverse engineering of data, IBM Syst. J. 37 (2) (1998) 246–269.
[3] Nicolas Anquetil, Timothy C. Lethbridge, Recovering software architecture from the names of source files, J. Softw. Maint. 11 (3) (1999) 201–221.
[4] Edmund C. Arranga, Ian Archbell, John Bradley, Pamela Coker, Ron Langer, Chuck Townsend, Mike Wheatley, In Cobol's defense, IEEE Softw. 17 (2) (2000) 70–72, 75.
[5] Gary Barnett, The future of the mainframe, Available via http://store.ovum.com/Product.asp?tnpid=&tnid=&pid=33702&cid=0, October 2005.
[6] Bert Kersten, Chris Verhoef, IT Portfolio management: a banker's perspective on IT, Cutter IT J. 16 (4) (2003) 34–40.
[7] Michael Blaha, A Manager's Guide to Database Technology: Building and Purchasing Better Applications, Prentice Hall, 2000.
[8] Michael Blaha, Patterns of Data Modeling, CRC Press, 2010.
[9] Nicholas G. Carr, IT doesn't matter, Harv. Bus. Rev. 81 (5) (May 2003) 41–49.
[10] Computerworld, MIPS management at mainframe organizations, August 2007.
[11] Compuware, Reduce MIPS. Save money, Presentation available via http://www.compuware-insight.com/pdfs/0307/Enterprise-MIPSManagementbrochure.pdf.
[12] Compuware, Save money on mainframe hardware and software: five simple steps to start managing MIPS, September 2008.
[13] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, The MIT Press, Cambridge, Massachusetts London, England, 1990.
[14] J. Chris, Database in Depth – Relational Theory for Practitioners, O'Reilly, 2005.
[15] J. Chris, Database Design and Relational Theory – Normal Forms and All that Jazz, O'Reilly, 2012.
[16] Chris J. Date, SQL and Relational Theory – How to Write Accurate SQL Code, Second Edition, Theory in Practice, O'Reilly, 2012.
[17] Kathi Hogshead Davis, Lessons learned in data reverse engineering, in: WCRE, 2001, pp. 323–327.
[18] Virginie Detienne, Jean-Luc Hainaut, Case tool support for temporal database design, in: ER, 2001, pp. 208–224.
[19] Mike Ebbers, Wayne O'Brian, Bill Oden, Introduction to the New Mainframe: z/OS Basics, IBM's International Technical Support Organization, July 2006.
[20] Scott Fagen, David Luft, Optimizing mainframe success by simplifying mainframe ownership, Available via https://ca.com/Files/TechnologyBriefs/optimizing-mainframe-success-tb.pdf.
[21] Antonio Jose Ferreira, Performance tuning service for ibm mainframe, Available via http://www.isys-software.com/TuningService(ajf).pps.
[22] Mark Fontecchio, CA aiming to ease mainframe software licensing costs, Available via http://searchdatacenter.techtarget.com/news/article/0,289142,sid80_gci1295640,00.html, January 2008.
[23] E.R. Gansner, E. Koutsofios, S.C. North, K.-P. Vo, A technique for drawing directed graphs, IEEE Trans. Softw. Eng. 19 (3) (1993) 214–230.
[24] Joris Van Geet, Serge Demeyer, Lightweight visualisations of COBOL code for supporting migration to SOA, ECEASST 8 (2007), http://eceasst.cs.tu-berlin.de/index.php/eceasst/article/view/117.
[25] Francesco Guerrera, Citigroup looks to slash tech costs. FT.com, Financial Times, Available via http://www.ft.com/cms/s/0/24808782-462b-11de-803f-00144feabdc0.html, May 2009.
[26] Jean-Luc Hainaut, Vincent Englebert, Jean Henrard, Jean-Marc Hick, Didier Roland, Database reverse engineering: from requirements to care tools, Autom. Softw. Eng. 3 (1/2) (1996) 9–45.
[27] Jean Henrard, Didier Roland, Anthony Cleve, Jean-Luc Hainaut, An industrial experience report on legacy data-intensive system migration, in: ICSM, 2007, pp. 473–476.
[28] Jean Henrard, Didier Roland, Anthony Cleve, Jean-Luc Hainaut, Large-scale data reengineering: return from experience, in: WCRE, 2008, pp. 305–308.
[29] IBM, IBM – DB2 express-C, Available via http://www-01.ibm.com/software/data/db2/express/.
[30] IBM, SQL Reference, volume 1: Select-Statement, Available via http://publib.boulder.ibm.com/infocenter/db2luw/v8/index.jsp?topic=/com.ibm.db2.udb.doc/admin/r0000879.htm.
[31] IBM, Enterprise COBOL for z/OS compiler and runtime migration guide, Technical report, International Business Machines Corporation, 2007, Available via http://publib.boulder.ibm.com/infocenter/pdthelp/v1r1/index.jsp?topic=/com.ibm.entcobol.doc_4.1/MG/igymch1025.htm.
[32] IBM, DB2 universal database for z/OS version 8 – administration guide, Available via, http://publib.boulder.ibm.com/infocenter/dzichelp/v2r2/topic/com.ibm.db2.doc.admin/dsnagj19.pdf?noframes=true, June 2009.
[33] IBM, DB2 version 9.1 for z/OS – SQL reference, Technical report, International Business Machines Corporation, May 2009, Available via http://publib.boulder.ibm.com/infocenter/dzichelp/v2r2/topic/com.ibm.db29.doc.sqlref/dsnsqk16.pdf?noframes=true.
[34] IBM, DB2 version 9.1 for z/OS, application programming and SQL guide, Technical report, International Business Machines Corporation, June 2009, Available via http://publib.boulder.ibm.com/epubs/pdf/dsnapk14.pdf.
[35] IBM, DB2 version 9.1 for z/OS: performance monitoring and tuning guide, Technical report, International Business Machines Corporation, October 2009, Available via http://publib.boulder.ibm.com/infocenter/dzichelp/v2r2/topic/com.ibm.db29.doc.perf/dsnpfk16.pdf?noframes=true.
[36] IBM, Enterprise COBOL for z/OS V4.2 language reference, Technical report, International Business Machines Corporation, August 2009.
[37] IBM, IBM DB2 universal database – SQL reference version 8, Technical report, International Business Machines Corporation, June 2009, Available via http://publib.boulder.ibm.com/infocenter/dzichelp/v2r2/topic/com.ibm.db2.doc.sqlref/dsnsqj17.pdf?noframes=true.
[38] IBM, IBM Tivoli OMEGAMON XE for DB2 on z/OS, version 4.2.0, Technical report, International Business Machines Corporation, 2009, Available via http://publib.boulder.ibm.com/infocenter/tivihelp/v15r1/index.jsp?topic=/com.ibm.omegamon.xe_db2.doc/ko2rrd20228.htm.
[39] IBM, Introduction to DB2 for z/OS, Technical report, International Business Machines Corporation, May 2009, Available via http://publib.boulder.ibm.com/infocenter/dzichelp/v2r2/topic/com.ibm.db29.doc.intro/dsnitk13.pdf?noframes=true.
[40] IBM, SQL reference for cross-platform development, Technical report, International Business Machines Corporation, 2009, Available via ftp://ftp.software.ibm.com/ps/products/db2/info/xplatsql/pdf/en_US/cpsqlrv31.pdf.
[41] IBM, DB2 version 9.1 for z/OS: performance monitoring and tuning guide, Technical report, International Business Machines Corporation, March 2010, Available via http://publib.boulder.ibm.com/epubs/pdf/dsnpfk17.pdf.
[42] IBM, z/OS: resource measurement facility performance management guide, Technical report V1R12.0, International Business Machines Corporation, March 2010, Available via http://publibz.boulder.ibm.com/epubs/pdf/erbzpm90.pdf.

[43] IBM, IBM DB2 universal database – SQL reference version 7, Technical report, International Business Machines Corporation, db2sql. Available via ftp://ftp.software.ibm.com/ps/products/db2/info/vr7/pdf/letter/db2s0e71.pdf.
[44] Rick Kazman, Haruka Nakao, Masa Katahira, Practicing what is preached: 80-20 rules for strategic IV&V assessment, in: C. Verhoef, R. Kazman, E. Chikofsky (Eds.), IEEE EQUITY 2007: Postproceedings of the first IEEE Computer Society Conference on Exploring Quantifiable Information Technology Yields, IEEE Computer Society, Amsterdam, The Netherlands, 2007.
[45] Hans-Bernd Kittlaus, Peter N. Clough, Software Product Management and Pricing: Key Success Factors for Software Organizations, Springer, January 2009.
[46] A.S. Klusener, C. Verhoef, 9210: The zip code of another IT-soap, Softw. Qual. J. 12 (4) (2004) 297–309.
[47] Steven Klusener, Ralf Lammel, Chris Verhoef, Architectural modifications to deployed software, Sci. Comput. Program. 54 (2005) 143–211.
[48] Ralf Lämmel, Chris Verhoef, VS COBOL II grammar version 1.0.4, Technical report, Vrije Universiteit Amsterdam, 2002.
[49] Ted MacNeil, Don't be misled by MIPS, Available via http://www.ibmsystemsmag.com/mainframe/tipstechniques/systemsmanagement/Don-t-Be-Misled-By-MIPS/, November 2004.
[50] Ian Moor, An introduction to SQL and PostgreSQL, Available via http://www.doc.ic.ac.uk/lab/labman/postgresql/x101.html.
[51] H. Müller, K. Wong, S. Tilley, Understanding software systems using reverse engineering technology, in: The 62nd Congress of L'Association Canadienne Francaise pour l'Avancement des Sciences Proceedings, ACFAS, 1994, pp. 41–48.
[52] Andy Oppel, Robert Sheldon, SQL: A Beginner's Guide, 3rd Edition, McGraw-Hill Osborne, 2008.
[53] William J. Premerlani, Michael R. Blaha, An approach for reverse engineering of relational databases, Commun. ACM 37 (5) (1994) 42–49.
[54] Lee Siegmund, An intuitive approach to DB2 for z/OS SQL query tuning, IBM Syst. Mag. (November/December 2006), http://www.ibmsystemsmag.com.
[55] Lee Siegmund, DB2 for z/OS SQL query tuning, continued, IBM Syst. Mag. (January/February 2007), http://www.ibmsystemsmag.com.
[56] Arie van Deursen, Tobias Kuipers, Rapid system understanding: two COBOL case studies, in: IWPC'98, 1998, pp. 90–97.
[57] N. Veerman, Automated mass maintenance of a software portfolio, Sci. Comput. Program. 62 (3) (2006) 287–317.
[58] N. Veerman, E. Verhoeven, Cobol minefield detection, Softw. Pract. Exp. 36 (14) (2006) 1605–1642.
[59] Niels Veerman, Automated mass maintenance of software assets, PhD thesis, Vrije Universiteit, Amsterdam, The Netherlands, 2007.
[60] L. Wall, T. Christiansen, R.L. Schwartz, Programming Perl, 2nd edition, O'Reilly & Associates, Inc., 1996.
[61] L. Wall, R.L. Schwartz, Programming Perl, O'Reilly & Associates, Inc., 1991.
[62] Marcin Zukowski, Balancing vectorized query execution with bandwidth-optimized storage, PhD thesis, Universiteit van Amsterdam, ISBN 978-90-9024564-5, 2009.